MSX-C入門 上巻

桜田幸嗣 著

アスキー出版局

商　標

MS-DOS は米国マイクロソフト社の登録商標です.
MSX, MSX2, MSX2+, MSX-DOS, MSX-DOS2, MSX-C は株式会社アスキーの商標です.
UNIX オペレーティング・システムは, AT&T のベル研究所が開発し, AT&T がライセンスしています.

はじめに

コンピュータというものは，つくづくイケナイ機械だと思います．なぜって，コンピュータで遊んでいると，どうしてあんなにも時を忘れて熱中してしまうのでしょう．なにもゲームにハマっているわけではないのです．もちろん，それ(良質のゲーム)で，風薫る初夏の休日を無駄につぶすようなことだってあります．はい，否定はいたしません．

しかし，ここで言いたいのは，プログラミングという作業の危険なまでの面白さなのです．寝る前に，ほんのちょいと，睡眠薬がわりにプログラムを1本モノにしよう，などと考えたのが運の尽き．気が付けば夜は白々と明け始め，それに比例して目は真っ赤，ああ今日も徹夜してしまった…….しかし，そのむなしさの中でも，大脳左半球はプログラミングの興奮にうち震えている．この喜びを知るあなた，そんなあなたには，ぜひ本書を読んでいただきたい．そして共にC言語を語りあかしたいものです．

よく耳にするC言語の売り文句は，いわく構造化されたコンパイラであり，したがってプログラムは作りやすい上に実行スピードも速い，また効率的なマシンコードを生成できるためシステムプログラミングにも最適で，現にあのUNIXはCで書かれているんだぞお，などというところでしょうか(言ってる意味がわからなくても気にせぬように)．たしかにウソではありません．

しかし，我々MSXユーザーが，なぜMSX-Cを使うのかと問われたなら，その理由はこういった大層なものでは決してないでしょう．MSXのユーザーは，だいたいにおいて，コンピュータを遊びの対象と考えています．プログラミングとは仕事のために行うものではなく，あくまで自分の好奇心を満足させるために行うものと考えています．みなさんもそうでしょう？

そんな我々にとって，MSX-Cは，なによりもまずプログラミングというものの楽しさを教えてくれる道具であります．なぜMSX-Cを使うのか，その答えは簡単，それがとんでもなく面白いからの一言です．ソフトウェアサイエンスの世界には，多くの巧妙な考え方やアルゴリズムがキラ星のように

散在しています．こんなにもエレガントな方法があったのかっ！　そう認識した瞬間は，何者にも換えがたい喜びがあります．その境地への架け橋となるのが，ほかならぬC言語なのです．BASIC ?　そんなもの，C言語の魅力の前にはメじゃないですぜ(とはいえやっぱりBASICも好きなのですけれどね，私は)．

本書では上下巻の2冊を通して，このMSX-Cによるプログラミングの面白さをみなさんにお伝えできるように努力したつもりです．さあ，C言語を覚えましょう．そして，さらなるプログラミングの世界に身をゆだねましょう．徹夜明けの，あの黄金の太陽が，みなさんの頭上に輝かんことを！

平成元年　六月吉日　著者記す

■上下巻の構成

この『MSX-C入門』は，上下巻の2分冊で構成されています．各巻の内容は以下に示すとおりです．

●上　巻

MSX-C Ver.1.1／Ver.1.2を対象としたC言語の入門編です．ただしページ数の都合もあり，ここでC言語のすべてが語られているわけではありません．この上巻では，ほんとうに必要最低限と思われるCの文法を中心に内容をまとめてみました．実際のCプログラミングを行う上でのテクニックについては，下巻にはいってから実践的に身に付けていただきます．

●下　巻

MSXのグラフィック機能やサウンド機能を目一杯に使いこなすための実践編です．スクリーンモード1の画面を使ったパックマン風の迷路追いかけゲームと，スクリーンモード5の画面を使ったグラフィックもぐら叩きゲームの2本のプログラムをメインの柱に据えて，MSX-Cを使う上でのさまざまな手法を解説していきます．なお前者は旧タイプのMSXでも実行できますが，後者にはMSX2以降の機種が必要です．

目　次

4章 これがCのプログラムだ 53

5章 基本的なプログラミング 83

Cのプログラムは如何にして組み立てられるのか 113

数値データの扱い方 147

10章 ポインタと配列と文字列 211

11章 データの保存と利用 233

本文イラスト 近江佳奈

1章

MSX-C ノススメ

1 1 どうしてMSX-Cを使うのか

この章ではMSX-Cの持つメリットを他のプログラミング言語と比較しながら簡単にまとめてみました．MSX-Cは本当に役に立つのだろうか，そんな不安と期待をお持ちのみなさん，これを読んで安心して，おおらかな気持ちでMSX-Cの勉強をはじめてください．

そんなことどうでもいいから早くMSX-Cの使い方を教えてっ，というせっかちな方はこの章をとばして2章へどうぞ．

●手軽だが遅いBASIC

MSXのユーザーの大多数は，たぶんBASICによってプログラミングの世界(あるいは泥沼)への第一歩を踏み出すのだと思います．読者の皆さんもそうだったでしょう．筆者も例外ではありません．

このBASICというやつは，じつに便利なプログラミング言語です．プログラムは簡単に作れるうえに，計算はもちろん絵も描ければ音楽も演奏できる．MSXをいじるなら，BASICさえあれば，ほかになんにもいらないと思ったりします．でも，それは初めのうちだけで，ある程度BASICを使い込んでくるとだんだん限界も目につき不満がつのってきます．その筆頭が「BASICは遅い」ということでしょう．データベースでも，ゲームでも，BASICで自作したプログラムは市販の同様なものにくらべると反応がワンテンポどころかスリーテンポぐらい遅れ，どうにも使い心地がよくありません．

●マシン語は高速だが面倒

そんなとき，まず思い付くのはマシン語の利用です．なにしろマシン語で書かれたプログラムの実行スピードはすこぶる速い．BASICの数倍から十数倍にも達します．

しかし，その代わりマシン語の扱いはBASICよりはるかに面倒でもあり

ます．なぜって，マシン語というのは非常にコマカイのですね．たとえばBASICなら画面に線を引くのはLINE命令一発でOKですが，同じことをマシン語でプログラムしようとすると，やれ点の位置を計算してデータをどこそこに格納してと，つまらない手間ばかりかかります．おまけにマシン語は針の先ほどのミスがあっても暴走してしまうため，プログラミング中いつも注意力を砥ぎ澄ませておかなくてはなりません．マシン語はスピードアップの特効薬には違いありませんが，この薬は副作用が多すぎるのが玉にキズなのです．

●コンパイラは手軽で速い

そこで登場するのが，MSX-Cをはじめとするコンパイラです．コンパイラはマシン語以外のプログラムをオートマチックにマシン語に書き直してくれます．つまり高速なプログラムが労せずして手にはいるわけです．

参考までにBASICとMSX-Cのスピード比較結果を1つ紹介しましょう．ここでは四則演算や条件判断が適当に含まれた簡単な例ということで，友好数を探すプログラムを作ってみました．リスト1.1がBASICによるもの，リスト1.2はそのC言語版です．

次に示すのは，両者が友好数ペア(1184，1210)を見つけるまでの実測値です．MSX-CはBASICの7倍半のスピードでした．ちょっとしたものではないですか．

言　語	実行時間
BASIC	2727秒
MSX-C	356秒

表1.1　友好数ペア(1184，1210)を見つけるまでの時間

次に，それならばということで，今度はこのプログラムを手作業でマシン語に書き直してみると，記録は265秒に縮まりました．さすがのコンパイラもこれにはかないません．

やっぱりそうか，コンパイラなんてマシン語を使えない人間のための二流の代用品なんだろうって？　いいえ，実はこのマシン語プログラミングに筆

```
100 DEFINT A-Z
110 FOR J=2 TO 10000
120   N=J: GOSUB 180: IF S<=J THEN 150
130   N=S: GOSUB 180: IF S<>J THEN 150
140   PRINT J;N
150 NEXT J
160 END
170 '
180 S=0
190 FOR I=N/2 TO 1 STEP -1
200   IF (N MOD I)=0 THEN S=S+I
210 NEXT I
220 RETURN
```

友好数とは，そのすべての約数(ただし自分自身は除く)の和が，互いに相手の数と等しくなるような数のペアのことです．

リスト 1.1　「友好数さがし」プログラム(BASIC 版)

```
#include <stdio.h>

main()
{
    int  j, s, sum();

    for ( j = 2; j <= 10000; ++j ) {
        s = sum( j );
        if ( s > j && sum( s ) == j )
            printf( "%d %d¥n", j, s );
    }
}

sum( n )
int n;
{
    int  i, s;

    s = 0;
    for ( i = n/2; i >= 1; --i )
        if ( (n % i) == 0 ) s += i;
    return  s;
}
```

リスト 1.2　「友好数さがし」プログラム(C 言語版)

者は1時間以上もかかっています．それに比べてCによるプログラミング時間はほんの数分でした．プログラムの作成から実行までをトータルで見れば，結局はコンパイラのほうが何倍も得だということになります．コンパイラは大いにものの役に立つのです．

言　語	合計時間(プログラム時間＋実行時間)
BASIC 語	1時間弱(約10分＋2727秒)
MSX-C	10数分(約10分＋356秒)
マシン語	1時間以上(1時間以上＋265秒)

表 1.2　トータルで考えた所要時間

● MSX-C　対　MSX べーしっ君

さて，MSX-C コンパイラを使うとマシン語なみのスピードが簡単に実現できると述べましたが，それは MSX-C に限らず他のどんなコンパイラでも同じことではあります．

たとえば前掲のリスト 1.1 を BASIC コンパイラ「MSX べーしっ君」で実行してみたら，同じ友好数ペアを見つけるまでの時間は 258 秒でした．MSX-C より速いくらいです．げげッ，よく見たら筆者の書いたマシン語プログラムより速いではありませんか．

だったら，わざわざ MSX-C なんて持ち出してこないで，べーしっ君でいいじゃないの，と考える方もいるでしょう．たしかに筆者もこのコンパイラにはよくお世話になります．とくに，20～30 行で書けるくらいの小さなプログラムは，べーしっ君にかぎるとさえ思っています．

しかし，ある程度プログラムに本腰を入れて作るなら，やはり MSX-C を選ぶでしょう．なぜならプログラム全体の見通しのよさが全然違います．プログラムの規模が大きくなるほど，そのご利益(りやく)も大きく現れてきます．これは「MSX-C 対べーしっ君」というよりも C 言語と BASIC の持って生まれた性格の違いです．

● Cはプログラミングしやすい「構造化言語」である

そもそも，どうすれば楽をしてプログラムが作れるか，この問題はコンピュータの誕生このかた多くの人々の考察のまとでした．そして考え出されたのが，

①プログラムの構造化：

プログラムを部分品(モジュール)に分け，それを簡単な構造に従って組み立てる．

②データの構造化：

処理するデータをわかりやすい形にまとめて扱う．

という2つの方法でした．実際，この構造化方針に沿ってプログラムを作りはじめると，こんがらかったロジックが自然に避けられ，プログラミングは驚くほど楽になります．

C言語はこの構造化プログラミングの考えを取り入れて作られています．ですから普通にプログラミングしていれば，いつのまにかプログラムやデータがモジュール化され，構造化されていきます．C言語のプログラムの作りやすさの源はじつにこの点にあるわけです．それにひきかえBASICでは，かなり意識的に努力しないと構造化プログラミングはできません．

このあたり，今のところはピンとこないかもしれません．またいくら文章で説明されても，たぶんなかなか理解できないことでしょう．しかし実際にMSX-Cでプログラムを作りはじめてみれば，Cのパワーがきっと実感できると思います．

● ライバルはPascalか

どうも話がくだくだと長くなりました．簡単にいえば**「コンパイラ型の構造化言語」**は非常によろしいということです．だから，BASICでもマシン語でもベーしっ君でもなく，MSX-Cが一番なのです．そうではありませんかっ，皆さん！

しかし世の中油断できないもので，MSX-Cがそう主張すると今度は「俺だって同じだ」と別のプログラミング言語が名のりをあげます．そのライバルの名はPascalです．

同じコンパイラ型の構造化言語とはいえ，かたやCは実践派から，かたやPascalは理論派から出発した，かなり性格の異なるプログラミング言語です．ところが出会ってみれば，似たもの同士のソックリさんというわけでした．

あえて違いを探すなら，Cはプログラムの「作りやすさ」を目指し，Pascalはプログラムの「わかりやすさ」が第一だというところでしょう．たとえば，さきほどのリスト1.2の一部をPascalで書き直すと**リスト1.3**のようになります．リスト1.2と比べてみれば，Cはプログラムの簡潔さを重視したリスト，Pascalは文書性を重視したリストになっていることがわかると思います．

```
function sum( n : integer ) : integer;
var i, s : integer;
begin
    s := Ø;
    for i := (n div 2) downto 1 do
        if (n mod i) = Ø then
            s := s + i;
    sum := s
end;
```

リスト1.3　文書性は高いが簡潔さには欠けるPascalのプログラム

● MSX-Cはマシン語の力を取り入れやすい

CとPascalの性格の違いは，このようなプログラムの字づら以外にも，いろいろな面で現われてきます．なかでもマシン語の扱いに関しては両者の方針にはだいぶ差があります．

まずPascalのほうは，マシン語などという下々(しもじも)の言語とはいっさい関係を持たないことを前提にしています．せっかくコンパイラの導入によってプログラムがわかりやすくなったのに，そこにマシン語がしゃしゃりでてきては，もとのモクアミだという考えなのでしょう．

ところがCはマシン語を敬遠するどころか，これほどマシン語との相性(あいしょう)がよい言語もありません．ダイヤモンドに目がくらみ，高級言語のたましいを捨てたな，という声がどこからか聞こえてきそうです．

しかしハードウェアの機能を目いっぱい引き出すには，マシン語との共存が不可欠なのです．とくにMSXというコンピュータは，VDPによるグラフィックスや，サウンドジェネレータ，マウスなど，面白い周辺装置をいろいろぶらさげています．このようなマシンをいじるには，MSX-Cとマシン語の組み合わせこそ黄金のコンビネーションといってよいでしょう．下巻では，その具体例も紹介しますので期待していてください．

● MSX-Cはシステムプログラミングの楽しさを味わえる

最後にもう1つ．MSX-Cの作り出すプログラムには，UNIX流のリダイレクション機能やパイプ機能が自動的に付加されます(4章で解説)．これは，MSX-DOS　TOOLSに代表される「ちょっといいツール」がガンガン作れるということです．

ただ，ここで「だから何？　ああいうののどこがいいの？」と聞かれると困ってしまいます．こういうレスポンスを返す人は，便利なツールというものの有用性を認めていないのでしょうね．

まあ，その気持ちはわからなくもありません．ツールのたぐいはプログラマにとって最も役に立つのですが，じゃあそれで何をプログラミングするかといえば，またまた別のツールを作ってみたりする人間が多い．風は風を呼び，ツールはツールを呼んで，結局なんのためのプログラムか，ということになります．

しかし筆者はあえていいます．このような「プログラムのためのプログラミング」は，実用性はともかく非常に楽しいものです．いや，そういうと語弊があるかもしれないので弁解しておきますが，もちろんC言語は実用性だって十分です．でもそれ以上に「プログラマの知的オモチャ」として面白いんだからしょうがありません．弁解になってないか．でも，この楽しさが味わえるだけでもMSX-Cには大きな存在価値があると，ここに断言してしまいましょう．

●結論：だからMSX-C

というわけで，ここまで述べてきた話を整理してみると，「MSX-Cを使うとよい理由」は，次の4つになります．

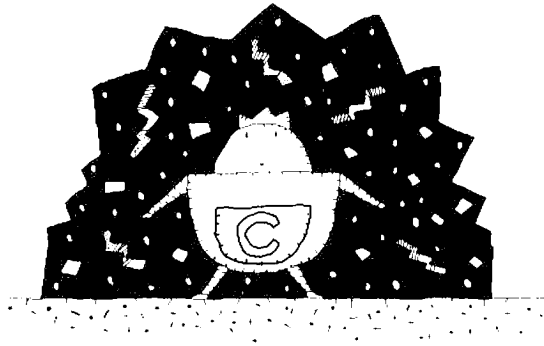
①コンパイラであるため，手軽な割りに高速性を発揮できる
②構造化言語であるため，大きなプログラムも作りやすい
③マシン語との相性がよいため，MSX のハードウェアを活用しやすい
④理屈はとにかく，システムプログラミングの楽しさが味わえる

以上を総括すれば，MSX-C は「一歩先を目指すプログラミング言語」といえるのではないかと思います．より速く，より大きく，よりマニアックなプログラム，それが MSX-C のターゲットです．

BASIC やマシン語に満足している方には，MSX-C は無用のプログラミング言語かもしれません．しかし，いままでの自分のプログラムに満足できない方，ぜひ MSX-C を使ってみてください．新しいプログラミングスタイルに，目が開かれる思いをすること請け合いです．

さて次の章では，MSX-C を使用する前の準備として，プログラミング環境の整備を行います．MSX のスイッチを入れる用意をしてお待ちください．

2章

プログラミングの環境を整える

MSX-C を手にして，初めて DIR コマンドでディスクの中身を見るとき，たいていの人が驚きます．こ，このファイルの多さはなんだあっ！

実は MSX-C のシステムディスクの中には，MSX-C コンパイラ本体に加えて，入門・中級者向けのサンプルプログラム，上級者向けのライブラリソースなどが混然一体となっているのでした．

MSX-C を使用するには，まず，このディスクから必要なファイルだけを抜き出し，若干のマシン語開発ツールと一緒にまとめて，プログラミング用のワークディスクを作ることから始めなくてはなりません．いわゆる「インストール」作業です．

本章では，この MSX-C のインストールの方法を最初から順を追って説明していきます．これには結構時間もかかりますが，気合いを入れて一気に片付けてしまいましょう．

2-1 これだけそろえば準備OK

まず必要なハードウェアとソフトウェアを準備しましょう．あとからアレがない，コレがないとあわてないように，以下のものを全部机の上に並べてから作業を始めてください．備えあればうれしいな，という金言もあることですしね．

・ハードウェア
① MSX本体
② ディスクドライブ(1台以上)
③ ディスプレイ

・ソフトウェア
① MSX-Cコンパイラ
② MSX-DOS　TOOLS

・MSX-Cインストール用のディスク
① 空ディスク(バックアップ用を含めて2枚あるとよい)
② ディスクラベル

● MSX本体

MSX-Cには，MSX-DOS1(以下DOS1と略)，あるいはMSX-DOS2(以下DOS2と略)の動作する，次のいずれかのマシンが必要です．

MSX1　――　64Kバイト以上のメインRAMを持つもの
MSX2　――　どの機種でも可
MSX2＋　――　どの機種でも可

●ディスクドライブ

ディスクドライブは，本体内蔵型でも外付け型でも1台あれば十分です．2台あるとインストール作業には便利ですが，実際にMSX-Cを使い始めると，どちらでも大差ありません．

●ディスプレイ

ディスプレイに関してぜいたくはいいません．家庭用TVで結構です．ただし，MSX2あるいはMSX2+を使用していて，なおかつ今後C言語をバリバリ使おうという心づもりがあるなら，1行80字表示が可能な専用モニターのほうがよいでしょう．そのほうが見やすくカッコよいプログラムが書けます．

● MSX-C コンパイラ

MSX-Cコンパイラには，現在のところ，Ver.1.1とVer.1.2の2種類のバージョンが存在しています．両者はそれぞれDOS1用のコンパイラとDOS2用のコンパイラという性格を持っていますが，Ver.1.1はDOS2で動作させることも可能です(ただしその場合,階層化ディレクトリをはじめとするDOS2特有の機能は，ほとんど活かせません)．

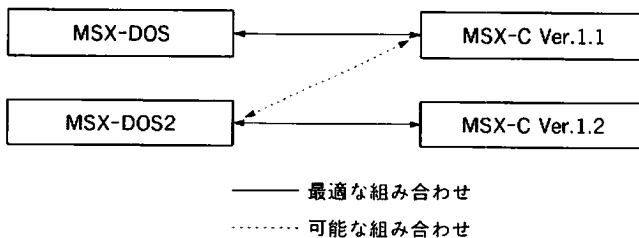



図2.1　MSX-Cコンパイラと MSX-DOSのバージョン対応関係

本書はどちらのバージョンのコンパイラをお使いの方でも読めるように書かれていますが，コンパイル操作などの実例を示す場合は，DOS1+Ver.1.1の環境下での実行画面を掲載しました．

● MSX-DOS TOOLS

MSX-Cを利用するためには，MSX-Cコンパイラ自体に加え，以下のプログラムツールが別途に必要です．

M80 アセンブラ
L80 リンクローダ
LIB80 ライブラリマネージャ
テキストエディタ

これらをそろえるには，すべてが1枚のディスクに収められた「MSX-DOS TOOLS」(以下TOOLSと略)を購入するのが一番てっとりばやいでしょう．本章でも，このTOOLSをお持ちになっていることを前提に，インストールの方法を説明していきます．

TOOLSには，このほかにもプログラミングの助けとなる便利なツールがいろいろとそろっていますから，ぜひ手に入れることをおすすめします．

●インストール用ディスクについて

必要なツール一式をそろえ，コンパイルの環境を整えるためには，やはりデータ容量の大きい2DDタイプのフロッピーディスクを使いたいものです．それでも1DDのフロッピーディスク1枚で間に合わないことはありません．必要なファイルをすべて1DDのディスクに集めても，百数十Kバイト程度の余地は残ります(プログラムを作っていくうちに，これくらいアッというまになくなりますけど)．

どちらにせよ，このディスクとは今後とも長いお付き合いをすることになります．できればまっさらの新品を用意してください．そしてラベルに「C言語学習用」と大書して，フロッピーディスクの上に貼り付けておきましょう．さあヤルぞっと心も引き締まります．

2-2 インストールの手順

必要な機材はすべて準備できましたか？　そうしたら，以下の手順でワークディスクを作成します．

なお，DOSとコンパイラのバージョンの組み合わせ方によって，作成するディスクの内容構成に若干の差がありますので，説明は3つに分けておきました．みなさんのお手元のシステムに合わせて，必要なところだけお読みください．

DOS1のためのインストール手順

DOS1＋Ver.1.1のシステムを使っている方は，以下の手順でインストールを行います．

① フォーマット済みのディスクを用意する

フォーマット作業はかならずDOS1で行ってください．異なるDOS（DOS2やMS-DOS）でフォーマットしたディスクを使うと，せっかくDOS1用の「MSXDOS.SYS」と「COMMAND.COM」をコピーしてもシステムは起動しません．ブートセクタに記録されるブートプログラムの内容が違ってくるからです．

② TOOLSのディスクから必要なファイルをコピーする

TOOLSの中の必要なファイルを以下に示します．なお，MED以外のエディタを利用されている方は，迷わずにそちらを使ってください．

MSXDOS.SYS	LIB80.COM
COMMAND.COM	MED.COM
M80.COM	MED.HLP
L80.COM	

③ MSX-C のマスターディスクから必要なファイルをコピーする

ここでは以下に示すファイルをコピーします．なお本書の後のほうでは，このほかのサンプルプログラムなども利用しますが，それらは必要になった時点で改めて指定させてもらいます．いまの時点では，この 13 個のファイルだけをコピーすれば結構です．

CF.COM	CK.REL
CG.COM	CLIB.REL
FPC.COM	CRUN.REL
LIB.TCO	CEND.REL
MX.COM	STDIO.H
AREL.BAT	BDOSFUNC.H
CREL.BAT	

```
MSXDOS   SYS     2432 85-08-23  9:29p
COMMAND  COM     6656 85-09-02 10:10p
M80      COM    20480 85-07-11  6:46p
L80      COM    10752 85-07-11  7:01p
LIB80    COM     4736 85-04-01 10:26p
MED      COM    15360 87-03-03 11:47p
MED      HLP     2196 87-02-09 11:27a
CF       COM    35072 87-08-12 12:00p
CG       COM    44800 87-09-29  1:07p
FPC      COM    28288 87-11-07  9:23p
LIB      TCO     2944 87-11-07  8:07p
MX       COM    11776 87-08-12 12:00p
AREL     BAT       24 87-08-12 12:00p
CREL     BAT       38 87-08-12 12:00p
CK       REL      128 87-08-12 12:00p
CLIB     REL    16000 87-11-07  8:07p
CRUN     REL     1664 87-08-12 12:00p
CEND     REL      128 87-08-12 12:00p
STDIO    H       3153 87-11-04  7:27p
BDOSFUNC H       5122 87-08-12 12:00p
      20 files     506880 bytes free
```

図 2.2 インストールを終了したディスクの内容

④ 追補：VRAM ディスクについて

これはオマケですが，128K の VRAM を持つ機種(MSX2／2+)を使用している方は，さらに Appendix2 の VRAM ディスクをインストールすると，快適な MSX-C ライフをおくることができます．

MSX-DOS は基本的にグラフィックスを使わないため，必要な VRAM はたかだか 6K バイトですから，MSX2 なら 100K バイト以上の VRAM が手付かずの状態で残っています．VRAM ディスクは，この部分を仮想ディスクとして利用するものです．これはフロッピーディスクよりずっと高速で，とくに MSX-DOS の外部コマンドが瞬時に実行できる快感は一度味わうとやみつきになるでしょう．

VRAM ディスクを利用するには，Appendix2 に示した手順に従って必要な 4 つのファイルを作り，ワークディスクに追加してください．ただしプログラムは 16 進ダンプリストの形で掲載してあります．これを打ち込むのは，かなりホネかもしれません．

DOS2＋Ver.1.1 のためのインストール手順

DOS2＋Ver.1.1 のシステムを使っている方は，ここに述べる手順でインストールを行います．

① フォーマット済みのディスクを用意する

フォーマット作業はかならず DOS2 で行ってください．DOS1 でフォーマットしたディスクは，いったん DOS2 上で FIXDISK コマンドにかけて正しいフォーマットに更正してから使う必要があります(FIXDISK コマンドの使い方は DOS2 のマニュアルを参照のこと)．

MSX-DOS 以外(MS-DOS など)でフォーマットしたディスクは，ブートセクタに記録されるブートプログラムの内容が異なるため使用できません．

② システムファイルをコピーする

DOS2 に付属してくるシステムディスクから，以下に示す 2 つのシステムファイルをコピーします．

MSXDOS2.SYS

COMMAND2.COM

③ 起動時実行バッチファイルを作る

システム起動時にMSX-Cの実行環境を整えるため，以下の2つのバッチファイルを作成します．

```
ramdisk 4064/d
copy command2.com h:¥
reboot %1
```

リスト2.1 AUTOEXEC.BAT

```
set temp=h:¥
set shell=h:¥command2.com
set append=%1¥files
path %1¥bin
```

リスト2.2 REBOOT.BAT

④ サブディレクトリを作る

必要なファイルを置くための，以下の3つのサブディレクトリを作ります．

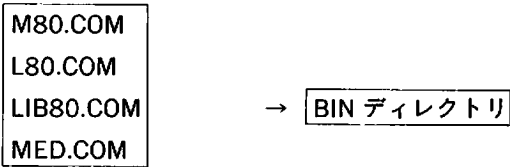
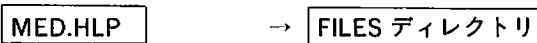
BIN	(コマンドを置く)
FILES	(読み出し専用ファイルを置く)
WORK	(作業用)

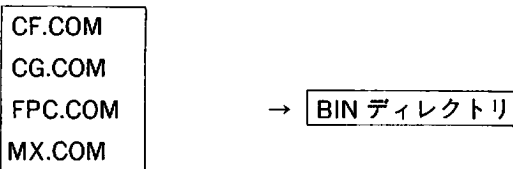
環境変数PATHによって指定されるBINディレクトリには，コマンドファイル(拡張子COMやBATを持つもの)を置きます．

環境変数APPENDによって指定されるFILESディレクトリには，各種の読み出し専用ファイル(インクルードファイル，ライブラリファイル，ヘルプファイル)を置きます．

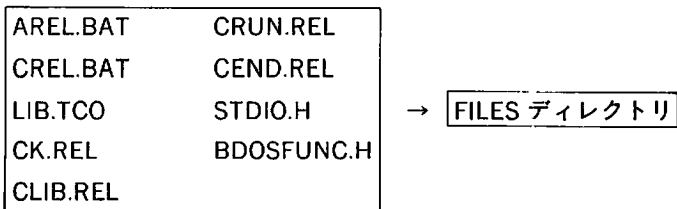
そしてWORKディレクトリで実際のコンパイル作業を行います．

⑤ TOOLS のディスクから必要なファイルをコピーする

以下のファイルを BIN ディレクトリの下にコピーします．

以下のファイルを FILES ディレクトリの下にコピーします．

⑥ MSX-C のオリジナルディスクから必要なファイルをコピーする

以下のファイルを BIN ディレクトリの下にコピーします．

以下のファイルを FILES ディレクトリの下にコピーします．

AREL.BAT	CRUN.REL
CREL.BAT	CEND.REL
LIB.TCO	STDIO.H
CK.REL	BDOSFUNC.H
CLIB.REL	

→ FILES ディレクトリ

「AREL.BAT」と「CREL.BAT」は拡張子 BAT を持っていますが，実際にはバッチファイルではなく MX コマンドの使うデータファイルです．こちらのディレクトリに置いても間違いではありません．

```
Volume in drive B: has no name
X-Directory of B:¥

MSXDOS2.SYS        4480
COMMAND2.COM      14976
AUTOEXEC.BAT         50
REBOOT.BAT           75
¥BIN
   M80.COM         20480
   L80.COM         10752
   LIB80.COM        4736
   MED.COM         15360
   CF.COM          35072
   CG.COM          44800
   FPC.COM         28288
   MX.COM          11776
¥FILES
   MED.HLP          2196
   AREL.BAT           24
   CREL.BAT           38
   LIB.TCO          2944
   CK.REL            128
   CLIB.REL        16000
   CRUN.REL         1664
   CEND.REL          128
   STDIO.H          3153
   BDOSFUNC.H       5122
¥WORK
217K in 22 files   480K free
```

図 2.3 インストールを終了したディスクの内容を XDIR コマンドで表示

DOS2+Ver.1.2のためのインストール手順

DOS2+Ver.1.2のシステムを使っている方は，ここに述べる手順でインストールを行ってください．

① フォーマット済みのディスクを用意する

フォーマット作業はかならずDOS2で行ってください．DOS1でフォーマットしたディスクは，いったんDOS2上でFIXDISKコマンドにかけて正しいフォーマットに更正してから使う必要があります(FIXDISKコマンドの使い方はDOS2のマニュアルを参照のこと)．

MSX-DOS以外(MS-DOSなど)でフォーマットしたディスクは，ブートセクタに記録されるブートプログラムの内容が異なるため使用できません．

② システムファイルをコピーする

DOS2に付属してくるシステムディスクから，以下に示す2つのシステムファイルをコピーします．

MSXDOS2.SYS
COMMAND2.COM

③ 起動時実行バッチファイルを作る

システム起動時にMSX-Cの実行環境を整えるため，以下の2つのバッチファイルを作成します．

```
ramdisk 4064/d
copy command2.com h:¥
reboot %1
```

リスト2.3　AUTOEXEC.BAT

```
set temp=h:¥
set shell=h:¥command2.com
set include=%1¥include
set append=%1¥lib
path %1¥bin
```

リスト 2.4 REBOOT.BAT

④ サブディレクトリを作る

必要なファイルを置くための, 以下の4つのサブディレクトリを作ります.

BIN	(コマンドを置く)
INCLUDE	(インクルードファイルを置く)
LIB	(ライブラリファイルを置く)
WORK	(作業用)

環境変数 PATH によって指定される BIN ディレクトリには, コマンドファイル(拡張子 COM や BAT を持つもの)を置きます.

環境変数 INCLUDE によって指定される INCLUDE ディレクトリには, MSX-C のヘッダファイル(拡張子 H を持つもの)を置きます.

環境変数 APPEND によって指定される LIB ディレクトリには, ライブラリファイルの他, いくつかの読み出し専用ファイルを置きます.

そして WORK ディレクトリで実際のコンパイル作業を行います.

⑤ TOOLS のディスクから必要なファイルをコピーする

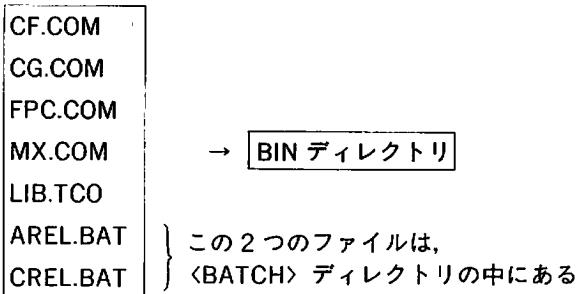
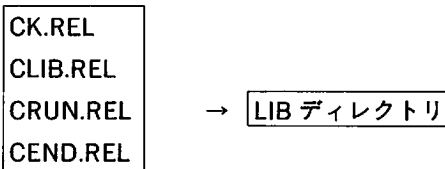
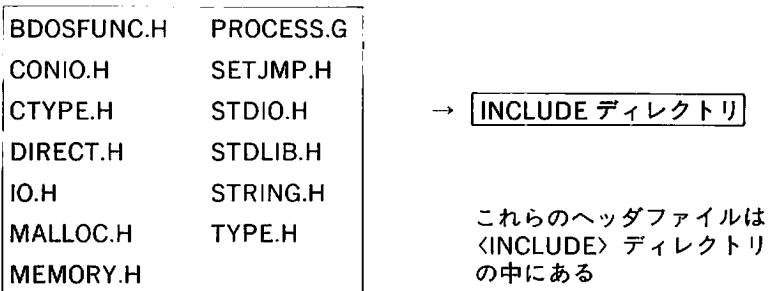
以下のファイルを BIN ディレクトリの下にコピーします.

M80.COM
L80.COM
LIB80.COM
MED.COM (DOS2 TOOLS の場合は KID.COM または AKID.COM)

MED を使う場合には以下のファイルを LIB ディレクトリの下にコピーします.

MED.HLP

⑥ MSX-C のマスターディスクから必要なファイルをコピーする

以下のファイルを BIN ディレクトリの下にコピーします．なお，「LIB.TCO」はコマンドファイルではなく，FPC コマンドが使うデータファイルですが，MSX-C コンパイラの Ver.1.2 では FPC と一緒のディレクトリに置くことになっています．また「AREL.BAT」と「CREL.BAT」も MX コマンドが使うデータファイルですが，MX と一緒のディレクトリに置くことになっています．

以下のファイルを LIB ディレクトリの下にコピーします．

以下のファイルを INCLUDE ディレクトリの下にコピーします．これはマスターディスクの INCLUDE ディレクトリの全内容です．

BDOSFUNC.H	PROCESS.G
CONIO.H	SETJMP.H
CTYPE.H	STDIO.H
DIRECT.H	STDLIB.H
IO.H	STRING.H
MALLOC.H	TYPE.H
MEMORY.H	

→ INCLUDE ディレクトリ

これらのヘッダファイルは
〈INCLUDE〉 ディレクトリ
の中にある

```
Volume in drive B: has no name
X-Directory of B:¥

MSXDOS2.SYS        4480
COMMAND2.COM      14976
AUTOEXEC.BAT         50
REBOOT.BAT           76
¥BIN
   M80.COM        20096
   L80.COM        10752
   LIB80.COM       4480
   KID.COM        28928
   AKID.COM       24576
   CF.COM         34816
   CG.COM         45056
   FPC.COM        26368
   MX.COM         10112
   LIB.TCO         4168
   AREL.BAT          23
   CREL.BAT          36
¥INCLUDE
   BDOSFUNC.H      8473
   CONIO.H          315
   CTYPE.H          922
   DIRECT.H         285
   IO.H             792
   MALLOC.H         420
   MEMORY.H         259
   PROCESS.H        289
   SETJMP.H         331
   STDIO.H         2517
   STDLIB.H         306
   STRING.H         424
   TYPE.H           588
¥LIB
   CK.REL           128
   CLIB.REL       17536
   CRUN.REL        1664
   CEND.REL         128
¥WORK
258K in 33 files   432K free
```

図 2.4 インストールを終了したディスクの内容を XDIR コマンドで表示

最後のチェック(全バージョン)

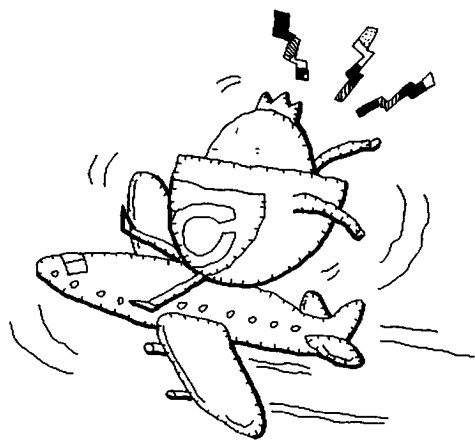
以上でMSX-Cの実行に必要な全ファイルの用意ができました．念のため，このワークディスクでMSX-DOSが起動できるかどうか確認しましょう．リセットスイッチを押すか電源を切ってもういちど入れ直し，うまくMSX-DOSが起動すれば，胸をなでおろしてください．

そして最後に，もう1枚フォーマット済みのディスクを用意し，これまで作ったディスクの内容をすべてコピーしておきましょう．このバックアップディスクは，大切なプログラムやデータを保存しておくために使用します．

MSX-Cでプログラミングしていると，BASICとは比較にならないほど，しょっちゅうディスクがアクセスされます．そのため，いつディスクが劣化して重要なファイルが読めなくなるかわかりません．そのような万一の場合に備えるのが，ここで用意したバックアップディスクの役割です．保存に値すると思うプログラムなどは，入力し終えたらすかさずバックアップを取っておく習慣をつけると安心です．

さて，やっとMSX-Cの利用体勢も整いました．昨今はやりの言葉でいえば，これでインフラストラクチャが完備したというわけです．次の章では，ここで作ったワークディスクを用いて，プログラムのコンパイルから実行までの方法を説明することにします．

3章

コンパイラの動作とその使い方

この章では、MSX-Cコンパイラが C のソースプログラムをどうやってマシン語まで持っていくのか、その処理の流れを見ていただきます.

MSX-C で我慢ならないのは、プログラムの実行まで何分も待たされるコンパイルのわずらわしさだ！　そんな不満を持つ方は、ここで紹介したコンパイル作業の内容を見てやってください. これじゃ時間がかかるのも無理はないと、少しは MSX-C に同情する気が起きるかもしれません.

なお、理屈はともかく早くコンパイラを動かしてみたいんだという方は、とりあえず本章の最後の「CC.BAT によるコンパイル」の項だけ目を通して、そのまま先へ進んでも結構です.

3-1 ソースプログラムの準備

ここでの第一の目的はコンパイル操作の練習です．そのためには練習台となる適当なCのプログラムが1つ必要です．

そんなプログラムがどこかに都合よくころがっていないか，と探してみたら，ありました，ありました．MSX-Cのオリジナルディスクは，サンプルプログラムの宝庫です．本章では，この中の「WC.C」を使って，コンパイルの練習をすることにします．

まずは，このサンプルプログラムをMSX-Cのオリジナルディスクからコピーしてください(図3.1)．

```
A>copy b:wc.c ⏎ ……………ドライブBのWC.CをドライブAにコピー
       1 file copied
A>dir wc ⏎      ……………正しくコピーできたか確認
WC       C       2223 87-Ø8-12 12:ØØp
       1 file    5Ø38Ø8 bytes free
```

注：MSX-C Ver.1.2では，サンプルプログラムWC.Cは<SAMPLE>というディレクトリに収められています

図3.1 WC.Cのコピー

このプログラムはファイルに含まれる行･文字･単語の数を調べるためのもので，WCという名前はワードカウント(WordCount)の頭文字をとってつけられています(WCといってもトイレではないんですね)．

ただし，本章ではまだプログラムの作り方を説明するわけではありませんから，プログラムの内容については理解できなくともかまいません．要するにバグがなくエラーを起こさないでコンパイルできれば，どんなプログラムでもよいのです．

MSX-C の 4 段活用

さて BASIC ならば，こうしてプログラムの用意ができたらすぐにも実行させられるのですが，C の場合はそう簡単にいきません．いったんこれをマシン語プログラムに変換しなければなりません．

このマシン語への変換作業のことを**コンパイル**といいます．そして，コンパイルされる前の C 言語のプログラムを一般に**ソースプログラム**，結果として作成されるマシン語プログラムを**オブジェクトプログラム**と呼びます．なお，C のソースプログラムには，「WC.C」のように必ず「.C」という拡張子を付ける約束になっています．

ところで一口にコンパイルといっても，MSX-C コンパイラは実は 1 個の独立したプログラムではありません．ソースプログラムを最終的なマシン語オブジェクトに変換するには，図 3.2 の 4 つのコマンドを順に実行する必要があります．

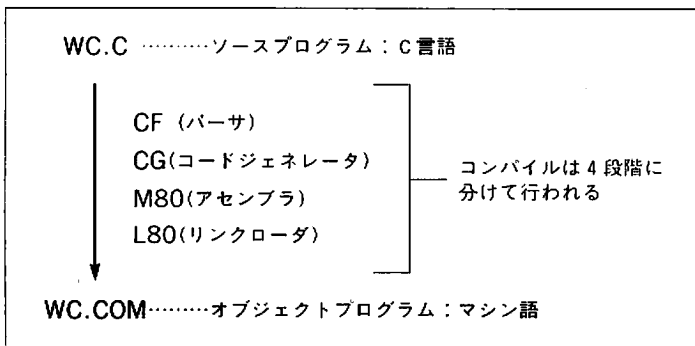



図 3.2 MSX-C の 4 段階の処理

通常のコンパイルでは，あとでも述べるように，この 4 つのコマンドはバッチ処理にまとめて実行されます．しかしここでは各コマンドの役割を理解するため，1 つずつ順番にその動作を追ってみましょう．

CF の動作

コンパイルの最初に実行するコマンドが CF です．CF はソースプログラムを解析して T コードと呼ばれる記号に変換し，その結果を「.TCO」という拡張子を持つファイルに書き出します．

● **CF を実行する**

それでは，用意した「WC.C」を図 3.3 のように CF でコンパイルしてください．このときファイル名に拡張子は付けず，ただ WC とだけ指定します．

ただし，これは MSX-C Ver.1.1 だけの制約で，Ver.1.2 からはファイル名に「.C」という拡張子を含めてもよいことになりました(拡張子を付けなければ，Ver.1.1 と同様に「.C」が指定されたものとみなします)．実は一般の C はみな拡張子も含めてファイルを指定する方法をとっていて，拡張子を省略する MSX-C Ver.1.1 のようなコンパイラは，ごくごく少数派なのです．

```
A>cf wc ↵ ……………CFの実行(ファイル名に拡張子は付けない)
MSX C ver 1.10P  (parser)
Copyright (C) 1987 by ASCII Corporation
complete
```

図 3.3 CF の実行

コマンドの実行が終わると，「WC.TCO」という T コードファイルが作られているはずです．これを DIR コマンドで確認してください(図 3.4)．

```
A>dir wc ↵
WC       C      2223 87-08-12 12:00p
WC       TCO    3200 89-04-10  1:00p ……………Tコードファイル
       2 files   499712 bytes free
```

図 3.4 T コードファイルの確認

● CF は何をする人ぞ

ここで，CF がどのような仕事をしているのか簡単に述べておきます．

CF はまず「# include」や「# pragma」など「#」ではじまる特殊命令を実行します．その機能はさまざまですが，実際のコンパイルの前処理という意味で，これらの実行は**プリプロセス**(pre-process)と呼ばれます(詳しくは各命令の説明を参照)．

次に CF は数字や名前を識別し，たとえば「ab = cde + fg；」という文なら，「ab」「=」「cde」「+」「fg」「；」という 6 語に分解します．この作業のことを字句解析，横文字で**レキシカルアナライズ**(lexical-analyze)といいます．

そして最後に，得られた単語の並び方から，この文は代入文であるとか，ここからここまでがループだとか，プログラムの意味の解釈を行います．これを構文解析，あるいは**パース**(parse)といいます．

つまり，CF は単に「パーサ」と呼ばれていますが，実際にはプリプロセッサとレキシカルアナライザとパーサ，この三役を務めているわけです．

● T コードをのぞいてみる

CF はこうして人間の書いたソースプログラムを解析し，コンピュータの理解しやすい T コード形式に変換して，コンパイラの次段に渡します．

この T コードはもとのソースプログラムより機械的に処理しやすいため，コンパイラだけではなく，MX(モジュールエクストラクタ)や FPC(ファンクションパラメータチェッカ)など各種ユーティリティプログラムでも利用され，MSX-C にとっては欠かせない存在となっています．

そこで T コードの中身はどうなっておるのだろうかと TYPE コマンドで表示してみると，すべての変数が 1 から始まる通し番号で管理されていたり，数式が逆ポーランド記法に変わっていたり，なかなか感心させられてしまいました．T コードの構造は公表されていないので詳しいところは不明でしたが，ヒマと興味のある方は解読してみてください．

CG の動作

CF の次に実行するコマンドは CG です．CG は T コードファイルを解析して，Z80 のアセンブリ言語に変換し，その結果を「.MAC」という拡張子を持つファイルに書き出します．

● **CG を実行する**

それではさきほど作られた「WC.TCO」を CG に与え，コンパイルの第 2 ステップを実行してください．このとき図 3.5 のように CG コマンドに与えるファイル名には，やはり拡張子は付けません（Ver.1.2 では拡張子を付けてもかまいません）．

図 3.5　CG の実行

CG コマンドは実行中いろいろな途中経過を報告してくれます．まず関数の処理を始めるとその名を表示し，その中で文を処理するごとにピリオドを表示します．コンパイルの進行状況がいつでも見えているおかげで，まだ終わんないのかコイツめ，というイライラも少しは解消するでしょう．

面白いのは，プログラムの最適化を行うごとにコロンが表示されることです．1 個のコロンが出てくるごとに，思わず MSX-C ガンバレと声をかけたくなります（そんなの筆者だけですかね）．

そして最後に complete と表示されれば，CG は無事終了です．

ここでは「WC.MAC」というファイルが作られているはずです．やはり DIR コマンドで確認してください(図 3.6)．

```
A>dir wc ↵
WC       C        2223 89-08-12 12:00p
WC       TCO      3200 89-04-10  1:00p
WC       MAC      5376 89-04-10  1:02p ……………アセンブリ言語プログラム
         3 files   493568 bytes free            が作られた
```

図 3.6　WC.MAC の確認

● CG の作るマシン語プログラム

この段階で作られるファイルは，アセンブリ言語で書かれたマシン語ソースプログラムです．Z80 のマシン語をご存じの方は「WC.MAC」を TYPE コマンドで表示してみてください．見覚えのあるマシン語命令が並んでいるでしょう(図 3.7)．マシン語プログラミングに慣れているなら，このプログラムの無駄な部分に手を入れて改善することもできます．

```
zerolo:
        push    hl
        inc     hl
        inc     hl
        ld      (hl),0
```

図 3.7　WC.MAC の一部を見る

ただし，「WC.MAC」に書き出されるのは，あくまでも元のソースプログラムに対応する部分だけです．文字列表示などの基本的なルーチンは，後述する「CLIB.REL」などのライブラリファイル中にリロケータブルモジュールとして記録されていて，ここでは出力されません．

M80の動作

CGの次に実行するコマンドがM80です．M80はマシン語ソースファイルをアセンブルし，「.REL」という拡張子を持つファイルを作ります．

● M80を実行する

それでは図3.8のように「WC.MAC」をアセンブルしてください．このときM80コマンドに特有の約束事として，ファイル名の前に「=」を付けることと，ファイル名の後ろに「/Z」を付けることを忘れてはいけません．なぜか？　という質問は発さないでください．M80というコマンドは，こう使うと決められているのです．

本書の範囲を超えるため，ここではM80に関する説明はひかえますが，MSX-DOS TOOLSのマニュアルには，M80の使い方の説明が詳しく解説されています．それを読めばM80の使い方もよくわかる(のではない)でしょう(か)．

```
A>m8Ø =wc/z ⏎ ……………M80の実行

No Fatal error(s)
```

図3.8　M80の実行

実行が終わると「WC.REL」というファイルが作られます．やはりDIRコマンドで確認してください(図3.9)．

```
A>dir wc ⏎
WC       C       2223 87-Ø8-12 12:ØØp
WC       TCD     32ØØ 89-Ø4-1Ø  1:ØØp
WC       MAC     5376 89-Ø4-1Ø  1:Ø2p
WC       REL     128Ø 89-Ø4-1Ø  1:Ø5p ……………リロケータブルモジュール
       4 files    49152Ø bytes free
```

図3.9　WC.RELの確認

●リロケータブルモジュールについて

M80の作るファイルは**リロケータブルモジュール**と呼ばれ，複数のモジュール同士を組み合わて簡単にマシン語プログラムが作れるように工夫されています．ですから，いろいろ便利なサブルーチンをリロケータブルモジュールの形でたくさん用意しておけば，あとはそれを積木のように組み立てるだけでプログラムが作れてしまいます．

MSX-Cには，入出力やファイル操作などの基本的なサブルーチンが，このリロケータブルモジュールの形でたくさん用意されています．いま作られた「WC.REL」も，それらのモジュールと一緒にまとめられて最終的なマシン語プログラムになっていくわけです．

リロケータブルモジュールというものは，MSX-Cが開発される前からマシン語プログラムの開発に使われていたのですが，こいつがあればこそ，MSX-Cもこの世に生まれ出づることができたのでしょう．リロケータブルモジュールとは，それくらい重要なアイテムなのです．

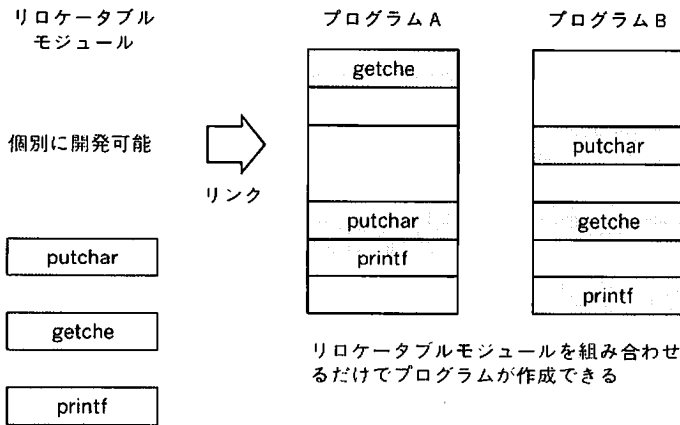



図3.10　モジュールごとのプログラム開発

ところでリロケータブルモジュールはTコードファイルやアセンブリ言語ファイルとは違い，TYPEコマンドで内容を見ることができません．また，データがビット単位で詰め込まれているため，1バイトごとに16進表示して

みても，何が記録されているのか判別できません．そのためリロケータブルモジュールの中身を知るには，LIB80 コマンドを使わなければなりません．その具体的な方法は下巻で説明しましょう．

L80 の動作

いよいよ最終段階です．ここでは L80 を用いてリロケータブルモジュールをリンクし，実行可能なマシン語プログラムを作ります．完成したプログラムは「.COM」の拡張子を持つファイルに書き込まれ，MSX-DOS の外部コマンドになります．

● L80 を実行する

L80 の実行方法は図 3.11 に示すとおりです．指定するパラメータが多いので，間違いのないように入力してください．

```
        ┌─リンクするファイルの指定          ┌─WC.COMを作れという指定
A>l8Ø ck,wc,clib/s,crun/s,cend, wc/n/e ⏎

MSX.L-8Ø  1.ØØ  Ø1-Apr-Ø5  (c) 1981,1985 Microsoft

Data    Ø1Ø3    1C2B    < 6952>

3317Ø Bytes Free
[Ø1Ø3   22CD      34]
```

図 3.11 L80 の実行

ここで L80 に与えているパラメータは，「CK.REL」「WC.REL」「CLIB.REL」「CRUN.REL」「CEND.REL」の 5 ファイルをリンクし，「WC.COM」を作れということを意味しています．

これらのファイルの中で，「CLIB.REL」と「CRUN.REL」の 2 つは複数のモジュールをまとめたライブラリファイルです．L80 を実行するとき，ファイル名の後ろに「/S」スイッチを指定しておくと，そのファイルが必要なとき

に限りリンクさせるという芸当が可能ですが，さらにそれがライブラリファイルの場合は，そこから必要なモジュールだけを自動的に選び出してくれます．いやあ，L80ってホントに便利なツールですね．

L80によるリンクが終了すると，「WC.COM」が作られています．やはりこれもDIRコマンドで確認してください(図3.12)．

```
A>dir wc ⏎ ··············結果の確認
WC       C      2223 87-Ø8-12 12:ØØp
WC       TCO    32ØØ 89-Ø4-1Ø  1:ØØp
WC       MAC    5376 89-Ø4-1Ø  1:Ø2p
WC       REL    128Ø 89-Ø4-1Ø  1:Ø5p
WC       COM    87Ø4 89-Ø4-1Ø  1:1Øp ··············WCコマンド完成
       5 files   4823Ø4 bytes free
```

図3.12 WC.COMの確認

さあこれでコンパイルは終了，WCコマンドが完成しました．前にも述べたとおり，このコマンドはファイルの中の行数・単語数・文字数をカウントするものです．ためしに「WC.C」自身の内容を調べてみましょう(図3.13)．

```
A>wc wc.c ⏎
      145      283      2Ø77 ··············WC.Cには145行，283単語，2077文字
                                            が含まれているということを表す
```

図3.13 完成したWCコマンドの実行

やった成功だ！ というわけで，手間をかけたわりに結果はささやかなものでしたが，どうやらコンパイルは成功したようです．

●モジュールたちのプロフィール

上の作業で，L80はターゲットの「WC.REL」以外に4つのファイルをリンクしました．これらはそれぞれ以下に述べるような役割を持っています．この4つは，どんなプログラムをコンパイルするときでも，かならずリンクしてやる必要があります．

① CK.REL —— カーネル起動プログラム

カーネルとは，コマンドに与えられたパラメータを受け取ったり，リダイレクトやパイプ処理の準備を整えて，プログラム本体(main 関数)を実行するルーチンのことです．そして，そのカーネルを起動するのが「CK.REL」です．なおカーネル自体は次に述べる「CLIB.REL」の中に格納されていて，そのリンク時に取り込まれます．

② CLIB.REL —— 基本関数ライブラリ

これは各種の基本的な関数のモジュールを集めたライブラリです．リンクするときに「/S」という指定を付けておけば，L80 はこの中から必要なモジュールだけを選択します．

たとえば「CK.REL」と「WC.REL」をリンクすると，その段階でカーネルや文字列表示ルーチンが必要とされていることがわかります．そこで L80 は，それらを含んだモジュールを「CLIB.REL」から抜き出すというわけです．

③ CRUN.REL —— ランタイムルーチン・ライブラリ

これは乗除算や数値の比較など，プログラムの部品となる小さなサブルーチン(いわゆるランタイムルーチン)を集めたライブラリです．「CLIB.REL」と同様に，この中の必要モジュールだけがリンクされます．

「CRUN.REL」と「CLIB.REL」は，1つのライブラリにまとめても問題はないのですが，一人前の関数とランタイムルーチンのケジメを付けるため，MSX-C では別々にまとめられているようです．

④ CEND.REL —— プログラムエンドの目印

このモジュールは他のすべてのモジュールの後ろにリンクされ，プログラムの最終アドレスの目印になります．作業用のメモリ領域を必要とするプログラムなどでは，この目印を利用して空きメモリの先頭アドレスを知ります．

3-2 CCコマンドによるコンパイル

以上，MSX-Cによるコンパイルの手順を紹介してきました．しかしコンパイルのたびにこのような作業を進めるのは，いくらなんでも手間がかかりすぎるため，通常はこれらをバッチコマンドにまとめて実行します．ついでにバッチコマンドの最後で，コンパイラが作った中間生成ファイルを消してしまえば，ディスクもスッキリするでしょう．

実はMSX-Cのオリジナルディスクには，そのための「C.BAT」というバッチファイルがすでに存在しているのですが，残念ながら「C.BAT」のコンパイル手順にはここまでの説明と少し異なる部分があります．そこで本書では新しく「CC.BAT」というバッチファイルを作ることにしました．

「CC.BAT」の作成

リスト3.1が「CC.BAT」の内容です．スクリーンエディタなどを使って入力してください．

```
cf %1
cg %1
m8Ø =%1/z
del %1.tco
18Ø ck,%1,clib/s,crun/s,cend, %1/n/e
del %1.mac
del %1.rel
```

リスト3.1 「CC.BAT」 —— コンパイラの元締め役

この中で，L80コマンドの実行前に「del %1.tco」を行ってTコードファイルを消去しているのは，図3.14に示すように，ディレクトリを表示したと

きソースファイルの直後にコマンドファイルが来るようにするための工夫です．A型人間はこういう部分についこだわります．

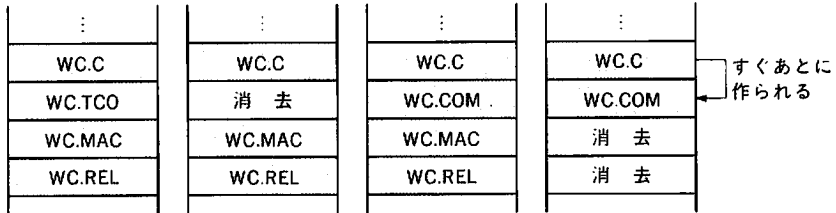
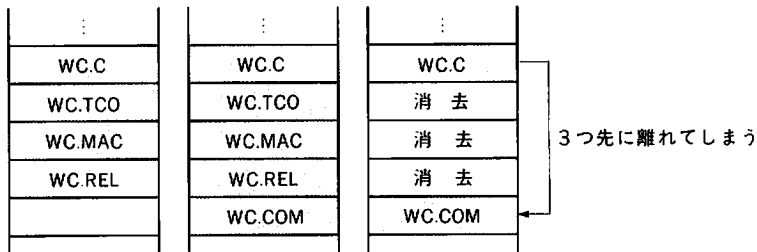



図 3.14 リンクの前にTコードファイルを消去する理由

また，RAMディスクやVRAMディスクが利用できるならば，そこに中間ファイルを作るとコンパイルのスピードが少し速くなります．たとえばRAMディスクがHドライブに設定されている場合は，リスト3.2に示すような「CC.BAT」を作成してください．

```
cf -oH %1
cg -oH H:%1
m80 =H:%1/z
l80 ck,H:%1,clib/s,crun/s,cend, %1/n/e
del H:%1.tco
del H:%1.mac
del H:%1.rel
```

リスト 3.2 RAMディスク利用時のCC.BAT

再び「WC.C」のコンパイル

それではCCコマンドを使って，再び「WC.C」をコンパイルしてみましょう．

CCコマンドを使用するには，次のように拡張子を除いたファイル名(ここではWC)をCCの後ろに指定します．

```
A>cc  wc
```

たったこれだけで，あとは「CC.BAT」がすべての作業を肩代りしてくれるのですから，バッチコマンドとは偉大なものです．ある友人の曰く，「ひとつのファイルはすべてを統べ，ひとつのファイルはすべてを見つけ，ひとつのファイルはすべてを捕えて，くらやみのなかにつなぎとめる．コマンド横たわるバッチ処理のなかに……」だとか．

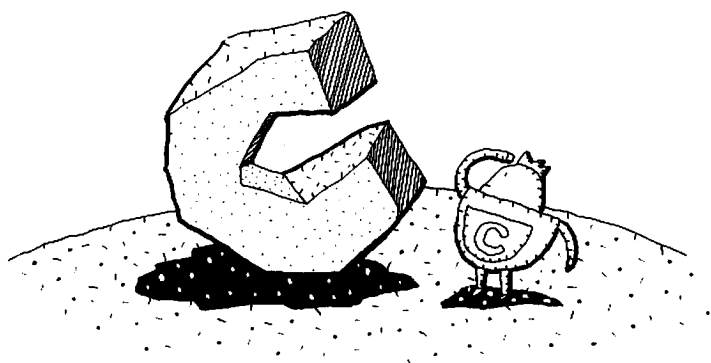
CCの実行が終了したら，「WC.COM」が作られ，中間ファイルは消されていること，および WC コマンドが前と同様に動作することを確認してください(図3.15)．

```
A>dir wc ⏎ ·············CC.BATで作られたWC.COMを確認する
WC       C      2223 87-08-12 12:00p
WC       COM    8704 89-04-10  1:15p ·············WCコマンドだけ残り，中間
        2 files   493568 bytes free              ファイルは消された
A>wc cc.bat ⏎
         7        17        112 ·············WCコマンドの動作を確認
```

図3.15　CC.BATの動作確認

コンパイルは成功しましたか？　うまく動作しなかった場合は「CC.BAT」の内容をもう1度チェックしてください．次の章からは，このCCコマンドを使ってプログラムのコンパイルを行っていきます．

4章

これがCの
プログラムだ

みなさんが初めてCのプログラムを見たとき，どんな感想をお持ちになりましたか．筆者の場合は，ムムッこいつはBASICとはだいぶ違うな，ということでした．まず見た目からして，プログラムは小文字で書かれているわ，行番号は存在しないわ，カッコのたぐいは妙に多いわ．

さらに，プログラムを実行するときの形態が，これまたBASICとCではかなり違います．BASICのプログラムは，まずRUNして，それからデータを入力して，というような「対話的」な環境で実行されるものでしたが，Cではそんな悠長な姿勢はハヤリません．コマンドラインで指定したパラメータをもって，ドカッとプログラムを実行してしまいます（前章で作ったWCコマンドがいい例です）．

このような言語を使うからには，BASICのときとはいささか発想も転換する必要があります．本章では，いくつかのサンプルプログラムを通して，まずはこういったC特有のプログラミングスタイルに慣れていただくことにしましょう．

4 1 Cのプログラムに触れてみる

さて前章で「CC.BAT」の用意も整い，Cで書かれたプログラムは自由にコンパイルできるようになりました．こうしてコンパイラが作り出したオブジェクトプログラムは，次のような使い方ができる点に大きな特徴があります．

①起動時にパラメータを指定できる．
②入出力のリダイレクションが行える．
③プログラムをパイプで何段も重ねられる．

そこで，まずは典型的なCのプログラムを例にとって，その使い方を体験してみることにしましょう．

プログラムの入力から実行まで

というわけで，また練習台となる適当なプログラムが必要になるのですが，MSX-Cのマスターディスクに収められているサンプルプログラムというやつは，前章で利用した「WC.C」も，それ以外のサンプルも，どれも出力結果がジミで，実行してもあまり面白味がありません．

そのため，ここでは入力の練習も兼ね，プログラムが簡単で，ちょっと見栄えのするプログラム「TRIANGLE.C」を用意しました．次のページにそのリストを示します(図4.1)．これを打ち込んでサンプルプログラムとして利用しましょう．

```
Ø: #include <stdio.h>
1:
2: putline(ch, n)
3: char  ch;
4: int   n;
5: {
6:     while (--n >= Ø)
7:         putchar(ch);
8: }
9:
1Ø: main(argc, argv)
11: int   argc;
12: char  *argv[ ];
13: {
14:     int   i;
15:     int   size, spc, len;
16:
17:     if (argc < 2)
18:         size = 5;
19:     else
2Ø:         size = atoi(argv[1]);
21:
22:     len = 1;
23:     spc = size;
24:     for (i = 1; i <= size; ++i) {
25:         putline(' ', spc);
26:         putline('+', len);
27:         putline(' ', spc-1);
28:         putchar('¥n');
29:         len += 2;
3Ø:         --spc;
31:     }
32: }
```

図 4.1　サンプルプログラム "TRIANGLE.C"

BASICでは，プログラムの入力も実行も，すべてBASICという1つの環境の中でまかなうことができました．それに対してCでは，プログラムの入力・コンパイル・実行の3つは，それぞれまったく別の操作になります．

●プログラムの入力

まずプログラムの入力には，テキストエディタと呼ばれる文字入力の専用ツールを使います．その入力方法は個々のエディタによっても異なりますが，

たとえばMSX-DOS TOOLSに収められているMEDというエディタを使う場合には，次のような手順をとります．

① テキストエディタ MEDの起動

MSX-DOSのコマンドラインから次のようにMEDを起動します．ここで指定している「TRIANGLE.C」は，作成したいファイルの名前です．すでに同じ名前のファイルが存在していれば，その内容が読み込まれてからエディタが起動します．

```
A>med triangle.c
```

② プログラムの入力

MEDを起動させたら，リストを見ながら，ひたすらプログラムを打ち込んでいきます．なお，これ以降本書ではリスト4.1のようにプログラムの行頭に参照用の行番号を付けて掲載しますが，これは実際のプログラムには関係ありませんから入力しないでください．

③プログラムのセーブとエディタの終了

プログラムをすべて入力し終えたら，それをディスクにセーブしてエディットを終了します．MEDでは，まず[ESC]を押すと画面の最下行に，

```
Command:
```

という表示が現れますから，そこで[q][⏎]の順にキーを押してください．すると，さらに，

```
Save (Y/N)?
```

と質問されます．ここで[y]を押すと，プログラムがセーブされ，エディタは終了します．

以上はMEDを使った場合のプログラム作成手順でした．ほかのエディタを利用している方は，それぞれの操作マニュアルをお読みのうえ，プログラムを入力してください．

●プログラムのコンパイル

プログラムを無事に入力し終えたら，CCコマンドによるコンパイル作業に移ります(図4.2)．ただし，前章のサンプルと違い，今回はみなさん自身の手で打ち込んだプログラムをコンパイルするわけですから，文字のタイプミス等によるエラーの発生がどうしても問題になるでしょう(これだけのサイズのプログラムを一発でノーミスコンパイルできたなら，あなたの今日の運勢は大吉です)．

不運にもコンパイル作業がエラーで中断してしまった場合は，再びエディタを起動して，プログラムの修正を行わなければなりません．次に述べるエラーへの対処法などを参考に，1つずつ誤りをツブしていってください．

図4.2 CCコマンドによるTRIANGLE.Cのコンパイル

●エラーへの対処法

コンパイルの途中でプログラムの間違いを見つけると，MSX-Cコンパイラはそのエラーの場所と種類を以下の形式のメッセージで知らせてくれます(これはCFコマンドが表示するメッセージですが，ほかのコマンドを実行するときにエラーが発生することはほとんどありません)．

```
line xxx column xxx: .........
      ↑          ↑    ↑
```

行の位置　文字の位置　エラーの種類

たとえば,「TRIANGLE.C」の7行目を,図4.3のように打ち間違えた場合を考えてみましょう.

```
   :
5: {
6:      while (--n >= 0)
7:          putchr(ch); ……………正しくはputcharである
8: }
   :
```

図4.3 スペリングの間違い

このプログラムをコンパイルすると,MSX-Cは図4.4に示すエラーメッセージを表示します.

```
A>cc triangle
        ⋮ 途中省略
line 7 column 14: undeclared function 'putchr'
errors detected
```

図4.4 エラーメッセージの例

このエラーメッセージは,プログラムリスト7行目の14文字目にエラーが見つかったということを表しています.ただしMSX-Cの出すエラーメッセージでは,行数も文字数も0から数え始めるので注意してください(それに合わせて本書のプログラムには0から始まる行番号を付けました).

ところで,図4.4のエラーは「未定義の関数putchrが使用された」という意味ですが,このメッセージからエラーの発生理由がスペリングミスだと見抜くことは,今の段階ではなかなか難しいかもしれません.ですから,もう少しC言語に慣れてプログラムの内容が理解できるようになるまでは,とにかくエラーの行番号を頼りに,リストを1文字ずつ調べて間違いの箇所を見つけるとよいでしょう.

もっとも,CではBASICと違って,エラーメッセージで示された行にエラーの直接の原因があるとは限りません.たとえば図4.5を見てください.

```
    :
3: char  ch ……………最後にセミコロンが抜けている
4: int   n;
5: {
    :
```

図4.5 セミコロンを打ち忘れた

これは実際には3行目の最後のセミコロンが落ちているのですが，コンパイラがそれをエラーと判定するのは，セミコロン以外のもの，すなわち4行目のintが現れた時点です．そのため，このプログラムをコンパイルすると，エラーの場所は4行目だと表示されてしまいます(図4.6)．

```
A>cc triangle
        ⋮ 途中省略
line 4 column Ø: ';' expected ……………4行目と表示されているが，実際の
errors detected                          エラーの原因は3行目にある
```

図4.6 エラーメッセージで示された行にエラーがない場合

このように指摘された行にエラーが見つからない場合は，プログラムリストを1行ずつ前にさかのぼってチェックしてください．たぶん近くの行にエラーの原因を発見できるはずです．

またCでは，エラーが1つ発見できてもそこで終わりとせずに，最後までコンパイルを実行し，見つかったすべてのエラーを表示します．そのため，たった1カ所の間違いが次々にエラーを誘発して，エラーメッセージの山が築かれてしまうこともあります．

そのような間違いの例を図4.7に示します．これは8行目の閉じブレース記号を閉じカッコとしてしまったものです．

```
5: {
6:     while (--n >= Ø)
7:         putchar(ch);
8: ) ……………正しくは閉じブレース記号「}」である
```

図 4.7 カッコの対応が正しくない

そして，図 4.7 をコンパイルしたときのエラーメッセージを図 4.8 に示します．本来のエラーは 8 行目の 1 個だけなのですが，ここでボタンが掛け違ったせいで，10 行目の解釈がおかしくなり，その次の行にも影響が及び，結局以降のプログラムはすべて不正なものとみなされてしまいました．

```
A>cc triangle
        ⋮途中省略
line 8 column Ø: bad element ……………本来のエラー
line 1Ø column 4: undeclared function 'main' ……これ以降は最初のエラーに
line 1Ø column 5: undeclared identifier 'argc'  誘発された副次的なエラー
        ⋮途中省略
line 2Ø column 2Ø: undeclared identifier 'argv'
line 34 column Ø: unexpected eof inside function 'putline'
```

図 4.8 副次的なエラーメッセージの山

このように，ちょっと考えらないほど大量のエラーメッセージが出た場合，それらはほとんど，1 個目のエラーが生んだ幻のエラーに違いありません．このようなときは，ともかく最初のエラーメッセージが示す間違いを訂正してみてください．それだけで，たぶんたくさんあったエラーの山は嘘のように消えてしまうでしょう．

●プログラムの実行

無事にコンパイルは終わりましたか？　そうしたら，いよいよ実行です．MSX-C で作ったオブジェクトプログラムは，MSX-DOS の外部コマンドとなり，その名前（ここでは TRIANGLE）を入力するだけで実行できます．

図4.9にTRIANGLEコマンドの実行例を示します．この例のとおりに三角形が表示されれば大成功．もしも表示がうまくいかなかった場合は，プログラムのどこかに入力ミスがあるはずですので，再びエディタを起動してリストをよ～くチェックしてください．

```
A>triangle ⏎
    +
   +++
  +++++
 +++++++
+++++++++

A>
```

図4.9　TRIANGLEコマンドの実行例

またCでは，コマンド名の後ろにパラメータを付けて細かい動作を指定できるようなプログラムが簡単に作れます．このサンプルプログラムでも，適当な数値を与えると三角形の大きさが変わるような工夫をしてみました．

たとえば図4.10の実行例では，コマンド実行時にパラメータ3を指定して，3段の三角形を描かせています．これ以外にもいろいろな数値を指定し，その結果を確かめてください．

```
A>triangle 3 ⏎
  +
 +++
+++++

A>
```

図4.10　パラメータを指定してTRIANGLEコマンドを実行

Cのプログラムの合体パワー

さらに，Cで作られたオブジェクトプログラムには，データの入力元や結果の出力先を変更するI/Oリダイレクションやパイプの機能が，たいへん効果的に利用できるという特徴があります．おかげで，Cでは複数のプログラムを組み合わせると，それぞれを単独で使うときの何倍ものパワーを発揮させられます．

●第2のプログラムの入力とコンパイル

このことを理解していただくため，もう1つのプログラム「DOUBLE.C」を登場させましょう(図4.11)．さきほどと同様，このプログラムもエディタで入力し，CCでコンパイルしてください．

```
 0: /*  1ｷﾞｮｳ ｦ ﾖｺ ﾆ 2ﾊﾞｲ ｽﾙ  */
 1:
 2: #include <stdio.h>
 3:
 4: main()
 5: {
 6:     char  line[80];
 7:
 8:     while (gets(line, 80)) {
 9:         line[strlen(line)-1] = '¥0';
10:         puts(line);
11:         puts(line);
12:         putchar('¥n');
13:     }
14: }
```

図4.11　サンプルプログラム“DOUBLE.C”

DOUBLEコマンドは，図4.12のように，キーボードから1行分のデータを入力するごとに，それを2つ横に並べて表示するという動作を行います．プログラムを終了させるには，行の先頭で[CTRL]＋[Z]([CTRL]キーを押しながら[Z]を押す)を入力します．

```
A>cc double ↵
      :
    コンパイル過程省略(無事にコンパイルできたものとします)
      :
A>double ↵
Test ↵          ..............Testと入力
TestTest        ..............Test Testと出力
abc ↵                         以下同様
abcabc                          :
(12345) ↵                       :
(12345)(12345)                  :
^Z              ..............[CTRL]+[Z]を入力すると終了

A>
```

図4.12 "DOUBLE.C"の実行

●リダイレクションとパイプの利用

さて，これでTRIANGLEとDOUBLEの2つのプログラムが手に入りました．BASICならば，プログラムを2つ集めてきても，それはプログラム2個ぶんの働きしかしないでしょう．しかしCのプログラムは，1+1が3にも4にもなる可能性を持っています．

実際に2つの機能を組み合わせるのは，プログラムの入出力リダイレクションとパイプです．これらは各コマンドを実行するときに，表4.1に示す記号を用いて実現されます．

機能	記号	意味
出力リダイレクション(新規書き込み形式)	出力コマンド>ファイル	・指定ファイルにデータを書き込む
出力リダイレクション(追加形式)	出力コマンド>>ファイル	・指定ファイルにデータを追加
入力リダイレクション	入力コマンド<ファイル	・指定ファイルからデータを読み出す
パイプ処理	出力コマンド｜入力コマンド	・次のコマンドにデータを受け渡す

表4.1 リダイレクションとパイプの記号

表4.1で「出力コマンド」と表したのは，画面に対して出力を行うプログラムのことで，TRIANGLEもDOUBLEも，どちらもそう呼ばれる資格を持っています．これらに対して出力リダイレクションを行うと，たとえば次のような処理が可能です．

triangle > x	xというファイルに三角形を書き込む
triangle >> x	xというファイルに三角形を追加する
double > y	キーボードから入力したデータを2倍してyというファイルに書き込む

「入力コマンド」のほうは，キーボードからの入力を受け取るプログラムのことで，ここではDOUBLEだけがそれに相当します．

double < x	xというファイルから読み込んだデータを2倍して画面に出力する

そして出力コマンドと入力コマンドをパイプ記号「｜」（SHIFT キーを押しながら ¥ を押す）で連結すると，両者の間でデータの自動的な受け渡しが行われます．

triangle ｜ double	三角形を2倍して画面に出力する

●組み合わせの妙を見よ

入力リダイレクション，出力リダイレクション，パイプの三者は，組み合わせて使うこともできます．図4.13にいくつか実例を示しました．

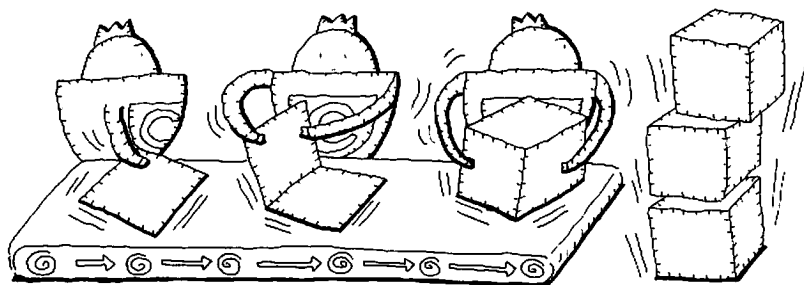
```
A>triangle 2 | double > tri  ··············TRIANGLEの出力をDOUBLEに渡して,
                                            その結果をTRIに書き込む
A>type tri
  +   +
 +++ +++

A>double < tri | double > tri  ·············TRIの内容をDOUBLEに読ませ，その
                                             出力結果をまたまたTRIに書き戻す
A>type tri
  +   +   +   +   +   +   +   +
 +++ +++ +++ +++ +++ +++ +++ +++

A>triangle 4 | double | double >> tri ···TRIANGLEの出力を2段のDOUBLEに通し,
                                          結果をTRIに追加書き込みする
A>type tri
  +   +   +   +   +   +   +   +
 +++ +++ +++ +++ +++ +++ +++ +++
    +       +       +       +
   +++     +++     +++     +++
  +++++   +++++   +++++   +++++
 +++++++ +++++++ +++++++ +++++++

A>
```

図 4.13 パイプとリダイレクションのさなざまな組み合わせ

いかがでしょう，こりゃなかなか面白そうだと思うでしょう？　こうしてプログラム同士を有機的に結び付けられる点が，BASICにはなかったCの大きな特徴であり，強みなのであります．

4-2 Cのプログラムの読み方

Cのプログラムがどのように用いられるか，ひととおり説明も済みましたので，そろそろプログラムの中身に話を移しましょう．

ただし，プログラムの個々の命令の意味については，まだ何も説明していませんからそっちに置いとくことにして，ここではごく形而下的なレベルでプログラムを眺めることにします．

プログラムの外見を拝見

というわけで，56ページの「TRIANGLE.C」をもういちどご覧ください（エディタを起動して画面にリストを表示させるとよろしい）．このプログラムを見て気がつくことをポンポンとあげてみましょう．

● Cのプログラムは小文字で書く

まず，Cではプログラム全体が小文字のアルファベットで書かれています．BASICでおなじみの条件判断文のキーワードifや，ループ文のキーワードforも，Cでは必ず小文字で書かないといけません．これをBASICのように大文字でIFやFORと書くとエラーになります．

変数などの名前に大文字を使用することは原則的には自由なのですが，Cのプログラミングの慣習として，やはりこれも小文字のみを使うことになっています．大文字を使うのは，こいつはちょっと普通じゃないのだぞ，ということを強調したいときに限ります．

● Cのプログラムには行番号が存在しない

BASICでは，プログラムを入力するにもGOTO文の飛び先を指定するにも行番号が不可欠でした．ところがCのプログラムからは，ご覧のとおり，

この行番号がすっかり姿を消しています．

なぜなら，Cではリストの入力にはテキストエディタを使いますし，プログラムの流れの制御にはGOTO文よりもっとわかりやすいwhile文やdo文などが用意され，行番号はまったく必要なくなったからです．

BASICにどっぷり浸かっている方は，行番号がなくなると，なにか自分の立脚点が失われてしまったような，アイデンティティ喪失の危機を感じるかもしれませんが，それは単に気のせいです．慣れればCの自由さのほうがずっと気にいるはずです(経験者は語る)．

●空白のバランスは個人の美的センスが決める

ところで，Cには行番号がないどころか，そもそも「行」というものに意味がありません．

実はCのプログラムはフリーフォーマットと呼ばれ，どこに空白を入れ，どこで改行をするかが，プログラマの判断に完全にまかされています．たとえば，リストの最初のほうに次のように書かれた部分がありますが……．

```
putline(ch, n)
char  ch;
int   n;
{
    while (--n >= Ø)
        putchar(ch);
}
```

スペース4個分あける

スペース8個分あける

図4.14　筆者がカッコイイと思うプログラムのフォーマット

あくまでもこれは筆者がかっこよいと思った書き方にすぎず，単語を1字1句この場所に置かねばならないという決まりはないのです．空白や改行がジャマくさいと感ずる方は，次のようなベタ詰めの「1行プログラム」を書くことも可能です．

```
                     ┌──────アルファベットから成る単語どうしの間には
                     │      スペースか改行が必要
                     ↓
putline(ch,n)char ch;int n;{while(--n>=Ø)putchar(ch);}
                       ↑
                       └────アルファベット以外のキャラクタが狭まっていれば
                            このように詰め込んでもかまわない
```

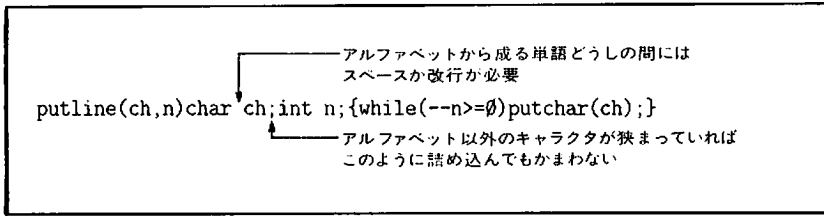



図 4.15　極端な例「ベタ詰め1行プログラム」

唯一の決まりは，「アルファベットから成る単語の間は，少なくとも1個のスペースまたは改行で区切る」ということ．つまり，

...charch ; intn...

のように単語を続けて書くのだけはルール違反です(アルファベット以外の記号ならスペースをあけずに続けて書いても大丈夫です)．

行の左端に空白をあけてインデント(字下げ)を付けるときなども，本書では紙面の大きさの関係もあってインデントの深さを基本的に4の倍数と決めていますが，みなさんがプログラムを入力するときはスペース3個でも5個でもいくつでもかまいません．

まあ要は，自分でわかりやすいと思う形に書くのがいちばんでしょう．いくつかプログラムを作りながら，みなさんも自分なりのプログラミングスタイルというものを見つけていってください．

●実行文はセミコロンで終える

Cのプログラムでは1行にいくらでも命令を詰め込めると述べましたが，その反対に，1つの文を複数行に分割することも可能です．たとえば，次のような長い文があるとしましょう．

```
x = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
```

Cではこれを次のように，文の途中で適当に改行を入れ，見やすく書くことができます．

```
x = 1 + 2 + 3 + 4
  + 5 + 6 + 7 + 8;
```

なぜ，この代入文が「x=1+2+3+4」で終わりにならず，「x=1+2+3+4+5+6+7+8」だと判断できるかというと，Cには文の最後は必ずセミコロンで終えるという決まりがあるからです．逆にいえば，Cの文はセミコロンが最後に付いていない限り，まだまだ先に続くものとみなされるのです．

ですからCでは，たった1つのセミコロンが，文の意味に大きな影響を与えます．文のどこにセミコロンをつけ，どこには必要ないかということは5章で説明しますが，とりあえずサンプルプログラムを入力するときには，セミコロンの有無にはよく気を付けてください．

関数がプログラムを組み立てる

さて，図4.1をざっと眺めると，このプログラムは大きく分けて2つの部分から成っていることがわかります．すなわち，

```
putline(ch,n)
char  ch;
int   n;
 {
    ......
 }
```

という前半のブロックと，

```
main(argc, argv)
int   argc;
char  *argv []
 {
    ......
 }
```

という後半のブロックです．厳密にいえば，両者の前にさらに「#include <stdio.h>」と書かれた行がありますが，これはプログラムの最初で必ず唱えるヒラケゴマの呪文のようなものと思って，いまのところは気にしないでください．

この2つのブロックでは，どちらも「関数の定義」を行っています．前者がputline()という関数の定義，そして後者がmain()という関数の定義です(本書では名前のうしろにカッコを付けて関数名を表します)．

●プログラミングの基本は関数の定義である

関数はCのプログラムの基本的な実行単位です．BASICのサブルーチンと似ていますが，名前で呼び出せること，引数をカッコに入れて渡せること，この2点においてCの関数はサブルーチンより何倍も便利です．

たとえば，プログラムの前半で定義しているputline()は，文字chをn個並べて表示する関数ですが，もしBASICで同様な機能のサブルーチンを作り，それを使って文字「A」を10個並べようと思ったら，たぶん次のようなコール命令を書くことになるでしょう．

```
CH$="A": N=10 : GOSUB 1000
```

一方，putline関数を使って「A」を10個並べるには，次のように書きます．

```
putline('A', 10) ;
```

こうすると'A'と10が自動的にputline()の使う変数chとnに代入されて，関数が実行されるのです．BASICに比べると，こちらの書き方のほうが圧倒的にスッキリしているでしょう？

さらに，BASICではサブルーチンを行番号で呼び出していたのに，Cでは「putline」という機能に沿った名前を使えるため，プログラムを読んだときに一段と理解しやすくなっています．

そして，いったん関数を定義すれば，これはもう新しい命令が出現したのと同じです．実際に，後半のmain()の中では，このputline()が次のように元からMSX-Cに用意されているputchar()と肩を並べてバシバシ使われています．

```
        :
  putline(' ', spc);
  putline('+', len);
  putline(' ', spc-1);
  putchar('¥n');
        :
```

いってみれば，Cのプログラミングとは，このような関数を作っては組み立てていくことの大いなる積み重ねなのです．

●ライブラリ関数はシステム御用達の関数である

こうして自分で定義する関数以外に，MSX-Cのシステムには初めから用意されている基本的な関数がいくつかあり，それらはライブラリ関数と呼ばれます．たとえば「TRIANGLE.C」にも登場したputchar()は，文字をひとつ表示するためのライブラリ関数です．

ただし，Cのライブラリ関数というものは，BASICの基本命令がそうであるような特別な存在ではありません．BASICでは，サブルーチンはいくら頑張ってもサブルーチンの身分のままで，PRINT命令やINPUT命令と同じレベルの命令にはなれませんでした．しかしCでは，自作の関数とライブラリ関数に本質的な差はまったくないのです．

実をいいますと，ライブラリ関数とは，かつてどこかの誰か(実際にはCの開発者)が作って「CLIB.REL」というライブラリファイルにストックしておいてくれた関数にすぎません．putline()もライブラリファイルに登録すれば立派なライブラリ関数に変身しますし，putchar()もライブラリファイルから削除してしまえばもはやライブラリ関数ではなくなります(こういったライブラリ操作は下巻のLIB80コマンドを使うと簡単に実行できます)．

● main関数からすべては始まる

Cのプログラムでは，これらのライブラリ関数や自作の関数が互いに呼び出したり呼び出されたりして作業を進めていきます．そして，ときにはかなり込み入った関係が作られることもあります．

その例として，図 4.16 に示した「SUMOMO.C」を見てください．

```
Ø: #include <stdio.h>
 1:
 2: mo_momo(), momo(), main(), mo(), sumomo();
 3:
 4: mo_momo()
 5: {
 6:     mo();  putchar(' ');  momo();
 7: }
 8:
 9: momo()
1Ø: {
11:     mo();  mo();  putchar(' ');
12: }
13:
14: main()
15: {
16:     sumomo();  mo_momo();
17:     momo();    mo_momo();
18: }
19:
2Ø: mo()
21: {
22:     putchar('ﾓ');
23: }
24:
25: sumomo()
26: {
27:     putchar('ｽ');  momo();
28: }
```

図 4.16 SUMOMO.C

ここでは，次のような 5 つの関数が定義されています．

mo_momo()	……「モ モモ 」と表示
momo()	……「モモ 」と表示
main()	……「スモモ モ モモ モモ モ モモ 」と表示
mo()	……「モ」と表示
sumomo()	……「スモモ 」と表示

だいぶ複雑ですが，どの関数がどの関数を呼び出しているか，わかりますか？　ためしに関数 mo_momo()の定義を見てみると……．

```
mo_momo()
 {
    mo() ;  putchar(' ') ;  momo() ;
 }
```

なるほど，この関数は mo()と putchar()と momo()の 3 つを呼び出しています．そこでさらに momo()の定義はどうなっているかというと……．

```
momo()
 {
    mo() ;  mo() ;  putchar(' ') ;
 }
```

ほほう，ここでもまた mo()と putchar()を利用しているぞ，などとチェックしていって，すべての関数の呼び出し／呼び出され関係をまとめると，図 4.17 のようなネットワークができあがります．Cのプログラムは，多かれ少なかれ，こういった関数呼び出しの連鎖で成り立っています．

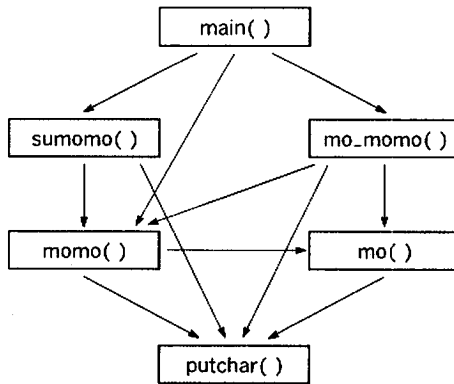



図 4.17　「SUMOMO.C」の関数の相関ネットワーク

さてそれでは，この「SUMOMO.C」をコンパイルして実行すると，どのような結果が得られるでしょう．みなさんも一緒に試してみてください．プログラムに誤りさえなければ，画面には次のメッセージが表示されるはずです．

スモモ　モ　モモ　モモ　モ　モモ

さあ，そこで疑問が浮かびます．上のように表示されたということは，すなわち main()が実行されたわけです．しかし，なぜ main()なのでしょうか？　図 4.16 のプログラムでは，合計 5 つの関数を定義しています．その中で，なぜ momo()でも sumomo()でもなく，main()が実行されるのでしょう？

実は，すべての関数の中で，たったひとつだけ特別な地位を与えられているものが，main()なのです．「main」という名前の関数を定義すると，それは自動的にプログラムの実行開始の入口とみなされます．BASIC のプログラムが必ず先頭の行から実行されるものと決められていたように，C では必ず main()からプログラムが実行されるのです．これは main()の定義がリスト中のどこにあっても関係ありません(普通は目立つようにプログラムの先頭か最後に置きます)．

逆にいえば，プログラムの中には必ず main()の定義を含める必要があります．そうでないとコンパイラはどこから手をつけていいのか判断できません．そのような(main()のない)プログラムを CC でコンパイルすると，L80 によるリンク作業の段階で，main()が定義されていない，というエラーメッセージが表示されてしまいます．

用意周到の宣言文

ところで，図 4.16 には，ここまで出てきた他のプログラムと少し違う点があります．すでにお気付きの方もいると思いますが，関数定義の本体が始まる前に，次のような文が書かれているのです．

```
mo_momo(), momo(), main(), mo(), sumomo();
```

こりゃなんだ？

●宣言文はコンパイラへの事前連絡である

タネを明かしてしまうと，この文は，これからmo_momo()，momo()，main()，mo()，sumomo()の5関数が使用されるぞ，という意味の「関数使用宣言」です．

もしこの宣言がなかったらどうでしょう．コンパイラは，コンパイル作業を進めながらプログラムリストを先頭の行から順に見ていきます．そして次のような関数の定義に出会います．

```
mo_momo()
 {
     mo();  putchar(' ');  momo();
 }
```

ところがコンパイラは，突然出てきたmo()やmomo()が何者であるかが理解できません．プログラムを書いた我々自身は，リストのうしろのほうで，これらがちゃんと関数として定義されていることを知っています．しかし，リストを先頭から見てきて今やっとここに到達したばかりのコンパイラは，後方にどんなことが書かれているのか，まったく知見を持ちあわせていないのです．

そこでコンパイラは，このmo()やmomo()を，「私の知らない関数だかナンだかが使用されている」と判断してエラーにしてしまいます．ためしに「SUMOMO.C」から関数使用宣言の部分を消してコンパイルを行ってみてください．コンパイラはエラーの山を吐き出してくるでしょう．

この問題に対する解決策は2つあります．1つは関数定義を並べる順番をうまく案配して，どんな関数も必ず「定義が先，使用は後」の方針を守ること．たとえば，本章の最初に紹介した「TRIANGLE.C」では，まずプログラムの前半でputline()という関数を定義し，後半でそれを使用していました．これならば，わざわざ関数の使用宣言などしなくとも，コンパイラは文句をいいません．

実をいえば，「SUMOMO.C」でも，関数定義を次の順におこなえば宣言は必要ありませんでした．こうすれば，どの関数も必ず使用される前にその定義が書かれていることになるからです．

mo()
　↓
momo()
　↓
mo_momo(), sumomo()　(どちらが先でもよい)
　↓
main()

しかし，いつでもそう都合よく順番に関数を並べられるとは限りません．たとえば，関数Aが関数Bを呼び出し，関数Bのほうも逆に関数Aを呼び出しているような場合は，どちらの関数定義を先に書いても，結局は一方が未定義のまま使われることになってしまいます．

そこで用いられるのが，もう1つの解決策である関数宣言です．これは，「後でこういう関数を使うつもりですので，そこんとこよろしく」と前もってコンパイラに挨拶しておき，万全の準備をさせようという方法です．「SUMOMO.C」の例のように，使用する関数を最初に宣言しておけば，それらがどんな順番でプログラム中に現れても，コンパイラは処理してくれるわけです．

なお，関数宣言は「この関数を使うかもしれない」という非常に弱い意味しか持っておりません．ですから宣言した関数を必ずしも実際に使う必要はありませんし，使わなければそれをプログラムする必要もありません．

その実例として，図4.18を見てください．

```
#include <stdio.h>

ahaha(), ufufu(); ............ 2つの関数を宣言しているが...

main()
{
    putchar('O'); }
    putchar('k'); } ............ どちらも使われていない
}
```

図4.18　関数宣言だけしておいて使わない例

ここでは最初にahaha()とufufu()の2関数を使用宣言しています．しかし結局これらの関数は使われもせず，定義もされず，単に名前が出たのみで終わってしまいました．これでもよいのです．関数の使用宣言というのは，イザというときの備えであって，必要なければ無視されるだけです．

● 「STDIO.H」はライブラリ関数の宣言をおこなう

さて，関数は定義が出てくる前に無断で使用するとエラーになる，ということを述べました．ここで問題となるのがライブラリ関数です．最前からサンプルプログラムの中でputchar()を繰り返し利用しましたが，その使用宣言はまったく行っていませんでした．ライブラリ関数というものは，コンパイラに断わりなく勝手に使ってもよいのでしょうか．

その答えは「No」です．前にもチラっと触れたとおり，Cではライブラリ関数とプログラマ自身が定義する関数の間に本質的な違いはなく，ライブラリ関数といえども，やはり無断で使用すればMSX-Cコンパイラは見逃してくれないのです．

ではなぜ，ここまで紹介したサンプルプログラムは無事にコンパイルできたのか．秘密はプログラムの先頭に置かれた次の命令にあります．

```
#include <stdio.h>
```

#includeは，プログラム中に別のファイルの内容を読み込む命令です．この命令に出会うと，コンパイラは山カッコ<>の中に指定された名前のファイルを読み込み，その内容を現在位置に挿入します(この動作をファイルのインクルードと呼びます)．

ですから，「#include <stdio.h>」という命令があると，そこに「STDIO.H」の内容が読み込まれてきます．そして，実はこの「STDIO.H」というファイルの中に，MSX-Cが用意したライブラリ関数の宣言がすべて書かれているのです．

言い換えれば，「#include <stdio.h>」という命令は，ライブラリ関数の使用宣言をズラズラ書き並べるのとまったく同じ効果を持つことになります．したがって，この命令さえプログラムの先頭に書いておけば，ここまで見てきたように，ライブラリ関数は自由に利用してかまわないわけです．

● Cでは変数も定義してから使用する

使用に先だって定義あるいは宣言を要するのは関数だけではありません．Cのプログラムでは，変数を使うにも，まずはコンパイラに対してそのことを知らせておく必要があります．

ここまで見てきたサンプルプログラム中にも，すでに変数定義の命令は登場しています．以下のように書かれている部分がみなそうです．

```
int  i;
int  size, spc, len;
char  ch;
char  line [80];
```

最初の「int i；」は整数を扱うint型の変数iの定義，その次はやはりint型の変数size, spc, lenを定義し，3番目では文字を扱うchar型の変数chを定義しています．最後の「char line [80]；」は少々特殊な形ですが，これは配列変数の定義です(変数の型や配列の扱いについては6章で詳しく説明します)．

これまで見てきたどのサンプルプログラムでも，変数を使う前には必ずこういった定義を行っていることに注意してください．もし定義なしに変数を使おうものなら，即座にエラーと判定されてしまいます．関数の場合もそうでしたが，Cコンパイラは素性の知れない存在に対して非常にキビシイのです．BASICに慣れていると，先生そりゃないよ，変数ぐらい勝手に使わせてよ，と思うでしょうが，こうした姿勢はバグの発生を未然に防ぐのに大きな効果があることを知ってください．たとえば，

```
int  ii;              ←変数iiの定義
ii = 1234;            ←変数iiを使用する
```

と書くべきところを，間違えて次のように打ち込んだとしましょう．

```
int  ii;
i1 = 1234;            ←iiを間違えてi1と書いてしまった
```

もしこれがBASICですと，そうか，新しく「i1」という変数を使うんだな

と解釈し（余計なお世話），プログラムを誤動作させてしまうでしょう．しかしCコンパイラならば，未定義の「i1」という変数が使われた，これはエラーに違いないと判断し，そのことをエラーメッセージで教えてくれます．

常に暴走の危険をはらむマシン語なるものを生成するCコンパイラは，こうしてプログラムを致命的なバグから守っているのです．

コメントは「/*」と「*/」で囲む

最後にコメント（注釈）の書き方を紹介して，Cのプログラミングスタイルに関する説明を終えることにします．

Cのプログラムでは，次のように「/*」（スラッシュ・アスタ）と「*/」（アスタ・スラッシュ）で囲まれた部分をコメントとみなします．

```
/* コメント */
```

必要ならばコメントは文の途中に入れることもできます．

```
if (energy < Ø) /* ﾀｲﾘｮｸ ｦ ﾂｶｲﾊﾀｼﾀ */  gameover=TRUE;
                            ↑
          厚かましくも文のドまん中に割り込んだコメント
```

図 4.19　文中にコメントを入れる

また，前に述べたとおり，Cではプログラム中に改行を入れることは自由ですから，次のように複数行に分かれていても，やはり「/*」と「*/」に囲まれた部分はすべてコメントとみなされます．

```
/*
   ｺﾚﾓ ｾﾞﾝﾌﾞ ｺﾒﾝﾄ         ……………もちろんコメントも複数行にまたがってかまわない
   ﾖﾛｼｲﾃﾞｽｶ(Y/N)?
*/
```

図 4.20　複数行にまたがるコメント

ちょっと凝って次のようなコメントを書いたプログラムを見かけることもあります．この場合は「/*」と「*/」をうまくデザインの一部として取り入れたわけです．

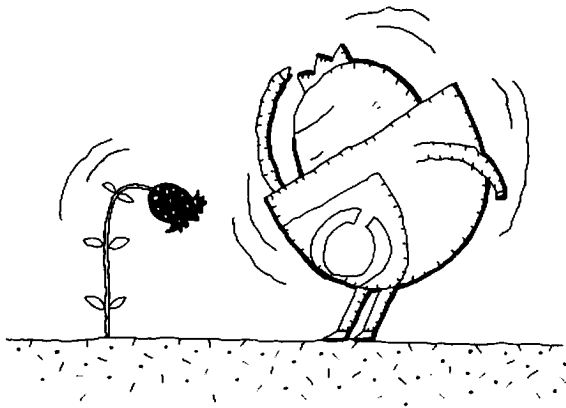
図4.21　デザインされたコメント

プログラムをデバッグするとき，必要ない命令を「/*」と「*/」で囲んで一時的にコンパイルを避ける「コメントアウト」の手法もよく用いられます．そのとき注意しなければならないのは，コメントのネスティング，つまり「/*」と「*/」で囲んだ中に，さらにコメントが書かれている次のような場合です．

図4.22　コメントのネスティング

MSX-Cでは，これは意図したとおりに①から④までがコメントとみなされます．しかしCの処理系によっては，①の次に最初に出て来る「*/」，すなわち③までをコメントとすることもあり，仕様が統一されていません．MSX-C以外のCを使うときには，この点に気を付けてください．

5章

基本的なプログラミング

いよいよ本章から実際のプログラミングの説明にはいります．ただし，これで前章のサンプルに出した「TRIANGLE.C」のようなプログラムが，すぐにも作れるのかなどと期待しないように．この章ではまず，データを入力して処理して出力するだけの，ごく基本的なプログラムの作成から始めましょう．そうして徐々に高度なテクニックへと進んでいくことにします．

前章で紹介したサンプルに匹敵するプログラムを自作するには，皆さんは本書の最後に達するまで，まだまだ何段階ものレベルアップを行う必要があります．プログラミングの道は奥深いのであります．

5 1 画面にデータを出力する

新しいプログラミング言語に接するとき，最初に覚えるべきものは何かというと，まあこの画面表示ということになるでしょう．なにしろ，コンピュータがどんなにスゴイ計算をしてくれても，その結果がこっちにわからなくては意味がありません．

これからＣを覚えようとしている皆さんにとっても，「プログラムをこう書けば結果はこうなる！」と目に見える形で答えが出たほうがプログラムの意味を理解しやすいでしょう．というわけで，プログラミングの第一歩は，画面に文字や数値を表示する方法の紹介から始めます．

1つの文字を表示する —— putchar()

MSX-Cに用意されている最も単純かつ基本的な画面表示の機能は，前章でも登場した**putchar()**による1文字出力です．

ある文字，たとえばaを表示するには，次のように文字の前後をシングルクォート記号で囲み，'a'という**文字定数**の形にしてputchar()に与えます．

```
#include <stdio.h>
main()
{
    putchar('a');
}

a ……………実行結果
```

図5.1　文字を表示する

このputchar()は，BASICのPRINT文と異なり，表示後の改行を行いません．そのため，putchar()を続けて実行すると，文字は次のように横に並んで表示されます．

```
#include <stdio.h>
main()
{
    putchar('M');
    putchar('S');
    putchar('X');
}

MSX ……………実行結果
```

図5.2 1文字出力を続けて行う

ここで登場した「文字定数」とは，文字1個を表すC特有の記法です．文字定数のシングルクォートの中には，たった1つの文字しか書けません．BASICには，この文字定数に相当するものはありません．

えっ？ BASICには文字列というものがあるじゃないかって？ いえいえ，文字列ならばCにもちゃんと用意されています．Cの文字定数は文字列とはまったく別の存在なのです(文字定数の驚くべき正体は，本章の後半で明らかになるでしょう)．

文字列を表示する —— puts()

さて，理屈の上からは1文字出力の機能さえあれば，画面表示に関してはもう何もコワイものはありません．どんな長いテキストを表示するのも，結局は1文字ずつの出力の積み重ねです．

とはいえ，ちょっとしたメッセージを表示するたびに，いちいちputchar()を書き並べるのも面倒な話ではあります．そこで必要は発明を産み，文字列を表示するためのライブラリ関数，puts()が用意されることになりました．

たとえば「C　loves　You」と表示したければ，次のプログラムのように，この言葉を文字列として puts()に与えます．C の文字列は，BASIC の文字列と同じく，前後をダブルクォートで囲んで表現します．

```
#include <stdio.h>
main()
{
    puts("C loves you");
}

C loves you ……………実行結果
```

図 5.3　文字列を表示する

この puts() も前の putchar() と同様に表示後の改行は行いません．ですから続けて実行すると，表示される文字列は次のように横一列につながってしまいます．BASIC の PRINT 文の感覚で使うと失敗するので御用心を．

```
#include <stdio.h>
main()
{
    puts("pot");
    puts("a");
    puts("to");
}

potato ……………実行結果
```

図 5.4　文字列表示を続けて行う

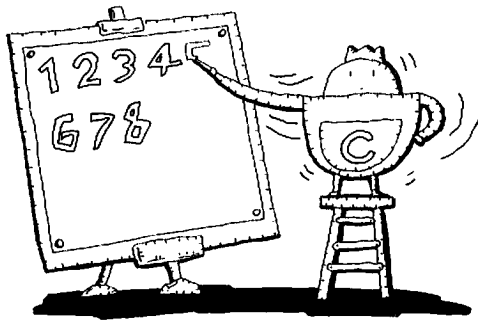
数値を表示する —— printf()

お次は数の表示です．数値の表示には，上の2つとはまた別のライブラリ関数，printf()が用意されています．この関数を使って，たとえば1234という数を画面に表示するには次のようなプログラムを書きます．printf()も表示後の改行は行いません．

```
#include <stdio.h>
main()
{
    printf("%d", 1234);
}

1234 ……………実行結果
```

図5.5　数値を表示する

さて，ここで不思議なものが現れました．カッコの中の"%d"とはなんぞや？　1234という数を表示するだけなのに，なぜ妙な記号を一緒に添える必要があるのか？

実はprintf()はたいへん多くの機能を持った関数なのです．数値を10進数として表示するだけでなく，16進数の形で表示したり，文字コードがその数値に相当する文字1個を表示することも可能です．そのためprintf()を使う場合は，そこで関数にどんな働きをさせたいかという情報も一緒に与えなければなりません．

“%d”もその1つで，これは「数値を10進数形式で表示せよ」という指定に当たります(10進数以外の表示方法は，これから段々に説明していきますので，しばらくの間お待ちください)．

改行を行う —— ニューライン文字

ここまで述べたように，putchar()やputs()やprintf()による表示は，通常は改行を行いません．改行が必要な場合は，¥nの記号で表される特別な文字を使用します．

¥nは「ニューライン文字」と呼ばれ，Cではこれを画面に出力することによって，改行を行う仕組みになっています．たとえばputs()で表示する文字列の最後に¥nを追加しておくと，次のように文字列ごとに改行して表示されます．

```
#include <stdio.h>
main()
{
    puts("ﾅﾅｷﾞ¥n");
    puts("ｳﾉ¥n");
    puts("ｻｸﾗ¥n");
}



ﾅﾅｷﾞ
ｳﾉ      ……………実行結果
ｻｸﾗ
```

図5.6 ニューライン文字で改行を行う

またニューライン文字は，文字列の最後だけではなく，先頭でも真ん中でもどこにでも自由に置くことができます．ですから，1つのputs()で，次のように複数行にわたる出力を行うことが可能です．

```
#include <stdio.h>
main()
{
    puts("ｲﾛﾊﾆﾎﾍﾄ¥nﾁﾘﾇﾙｦ¥n");
}

ｲﾛﾊﾆﾎﾍﾄ
ﾁﾘﾇﾙｦ              ……………実行結果
```

図5.7 ニューライン文字を文字列の途中に入れる

このニューライン文字は¥とnの2文字に分けることはできません．これは，あくまでも¥nと書いて1個の文字を表現しているのです．1個の文字でありますから，シングルクォートで囲んで'¥n'という文字定数を作ることもできますし，それを次のようにputchar()で出力すれば，やはり改行が行われます．

```
#include <stdio.h>
main()
{
    puts("ﾜｶﾖﾀﾚｿ");
    putchar('¥n');  ……………putchar( )を使って改行
    puts("ﾂﾈﾅﾗﾑ");
}


ﾜｶﾖﾀﾚｿ
ﾂﾈﾅﾗﾑ              ……………実行結果
```

図5.8 putchar()でニューライン文字を出力する

5.2 変数とデータの計算法

まとまったプログラムを書こうとすると，どうしても必要になるのが変数というやつです．変数とはナニモノかという点については，すでに BASIC でおなじみでしょうから，「それはデータを格納する入れ物である」などということは，この本ではもうあらためて説明はしませんが，かまいませんね．

文字変数の登場 —— char 型の変数

ということで，さっそくCにおける変数の使用例を紹介することにします．次に示したリストは，変数 c に代入した文字定数'X'を putchar()を使って表示するプログラムです．

```
#include <stdio.h>
main()
{
    char  c;

    c = 'X';
    putchar(c);
}

X  ……………実行結果
```

図 5.9　文字変数を使った文字表示

4章でも少し触れましたが，C では変数を使用する場合，そのことを前もってコンパイラに対して知らせておかなくてはなりません．上のプログラムでいいますと，次のように書かれた部分が変数 c の使用宣言です．

```
char    c;
```

ここで，宣言の最初に書かれているcharというキーワードは，変数cが文字データを扱うことを指定する型宣言です(charはcharacterの略)．このことからCでは文字データをchar型のデータとも呼びます．

なお，BASICの文字列変数は名前の後ろに＄記号を付けてC＄などと表しましたが，Cの変数にはそのような名前による型の区別はありません．char型の変数も，この次に紹介するint型の変数も，どれもみな同じ外見をしています．慣れないうちは，ちょっと戸惑(とまど)うことがあるかもしれません．

数値を扱う変数 —— int型の変数

文字データを扱うための変数がchar型であるのに対し，数値を扱うための変数はint型と呼ばれます(intはintegerの略)．int型の変数を使用する場合には，次のような宣言を行います．

```
int    i;
```

次のプログラムは，int型の変数に代入した数値をprintf()で表示する簡単なサンプルです．

```
#include <stdio.h>
main()
{
    int  i;

    i = 1024;
    printf("%d", i);
}

1024 ……………実行結果
```

図5.10　変数を使った数値の表示

Cのint型変数は,BASICの整数変数と同じく,−32768から32767までの整数を扱います.BASICの単精度実数や倍精度実数に相当する実数データはMSX-Cでは扱えません(一般のCでは単精度実数にあたるfloat型,倍精度実数にあたるdouble型が使えるのですが,MSX-Cにはそれらの型は用意されていません).しかし,もともとCは数値計算を行うためのプログラミング言語ではありませんから,さして不便に感ずることはないでしょう.

配列変数の使い方 —— Cの配列表現

多くのデータを一括して扱う場合に必要となるのが**配列変数**です.配列変数も,やはり使用する前に宣言をしておく必要があります.たとえば,int型の要素3個からなる配列aの宣言は,次のように行います.

```
int    a [3] ;
```

BASICでは,A(3)のように丸カッコで配列を表しましたが,Cの配列は,ブラケット記号を使ってa [3] と表現するところに注意してください.

次に示すのは配列を使った簡単なサンプルです.このプログラムは,a[0]とa [1] の積をa [2] に代入し,その結果をprintf()で画面に表示します.

```
#include <stdio.h>
main()
{
    int  a[3];

    a[0] = 39;
    a[1] = 42;
    a[2] = a[0] * a[1];
    printf("%d", a[2]);
}

15678 ...............実行結果
```

図5.11 配列変数を使う

ところで，このプログラムでは，最初に「int　a [3]；」という配列宣言を行ったにもかかわらず，実際にはa [0] からa [2] までの要素しか使用しませんでした．このことを不審に思ったかたはいませんか？

これはCの配列を使う上で非常に重要な注意点ですから，文字を太くして強調しておきましょう．**Cの配列では，宣言した大きさより1だけ少ない番号の要素までしか使えないのです．**

つまり，Cの「int　a [3]；」という配列宣言は，「a [3] までの要素を使う」という意味ではなく，「配列aは3個の要素を使用する」という宣言なのです．そして，Cの配列はBASICと同じく第0要素から始まりますから，最後の要素はa [3] ではなくa [2] になります．配列を使うプログラムでは，つねにこの点を忘れないでください．

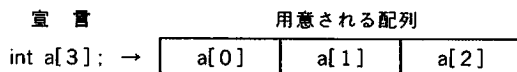



図5.12　配列宣言と要素数

CとBASICでは，2次元以上の配列を表現する方法も少し異なります．たとえばBASICでは，2次元配列を，

```
A(I, J)
```

のように表現しましたが，Cではこれを，

```
a [i] [j] ;
```

と表します．それぞれの添え字ごとに独立にブラケット記号で囲むわけです．3×3の大きさの配列を使用するならば，

```
int  a [3] [3];
```

と宣言することになります．この場合も，a [2] [2] までの要素しか利用できません．図5.13に2次元配列を使ったプログラムの実例を示します．

```
#include <stdio.h>
main()
{
    int  a[3][3]; ‥‥‥‥‥‥3×3の大きさの2次元配列の宣言

    a[0][0] = 0;   a[0][1] = 1;   a[0][2] = 2;
    a[1][0] = 1;   a[1][1] = 2;   a[1][2] = 3;
    a[2][0] = 2;   a[2][1] = 3;   a[2][2] = 4;

    puts("1 + 2 = ");
    printf("%d", a[1][2]);
}
```

図 5.13　2 次元配列を使う

さらに，3 次元の配列を使いたければ，もうひとつ添え字を追加して，

a [i] [j] [k]

と表現することになります．以下，4 次元配列，5 次元配列……，もすべて同様です．

加減乗除を行う —— C の算術演算記号

BASIC と同様に，C でも数値データに対して足し算・引き算・かけ算・割り算などの計算を行うことができます．C に用意された算術演算記号には，次の表に示すものがあります．

演算記号	意　味	BASICでの表現
x＋y	xとyを足す	X＋Y
x－y	xからyを引く	X－Y
x＊y	xとyを掛ける	X＊Y
x／y	xをyで割る	X／Y
x％y	xをyで割った余りを求める	X MOD Y
－x	xの正負符号を逆転する	－X

表 5.1　算術演算子一覧

これらはほとんどBASICの演算記号と一緒ですが，割り算の余りを求める剰余演算子だけは，C独自の記号を使うので注意してください．これはBASICでは「MOD」と書きますが，Cでは1文字の「%」という記号を用います．

またCでは，「−x」と書いて数値の符号を逆転することはできても，「+x」と書いてx自身を表すことはできません．つまり，

```
i = +123;
```

というような書き方はできないのです．+記号は2つの数値を加える用途にしか使えません．

次のプログラムは，加減乗除と剰余演算の簡単な使用例です．BASICをご存じの皆さんなら，このプログラムの意味はもう説明しなくともわかっていただけるでしょう．

```
#include <stdio.h>
main()
{
    puts("¥n 11 + 2 = ");    printf("%d", 11 + 2);
    puts("¥n 33 - 4 = ");    printf("%d", 33 - 4);
    puts("¥n 55 * 6 = ");    printf("%d", 55 * 6);
    puts("¥n 77 / 8 = ");    printf("%d", 77 / 8);
    puts("¥n 99 % 10 = ");   printf("%d", 99 % 10);
}

 11 + 2 = 13   実行結果
 33 - 4 = 29
 55 * 6 = 330
 77 / 8 = 9    ……………割り算の答は整数に切捨てられる
 99 % 10 = 9   ……………99÷10の余りは9
```

図5.14　算術演算子の簡単な使用例

ところで，Cのint型の変数はBASICの整数型変数と同じく−32768から32767までの範囲の数値を扱うということは前にも述べました．しかしCはBASICと比べると言語の性格がいくらかイーカゲンにできていて，この範囲を超える数値をint型の変数に代入してもエラーにはなりません．どうな

るかといえば，誤ったデータを代入したまま，平気で作業をどんどん先に進めてしまうのです．

たとえば次のプログラムを見てください．これは 1000×1000 を計算して答を変数 i に代入し，それを printf() で画面に表示するだけの簡単なプログラムですが……．

```
#include <stdio.h>
main()
{
    int  i;

    i = 1000 * 1000;
    printf("%d", i);
}

16960    ……………実行結果
```

図 5.15 代入値が int 型変数の範囲を越えてしまう例

この計算の答は 1000000 になるはずなのに，実行してみると結果はごらんのとおり，16960 などというワケのわからない値が表示されてしまいました．1000000 が int 型変数の範囲にはいりきらなかったのです．答が大きすぎて変数にはいらないならはいらないで，BASIC のように Overflow エラーのメッセージでも出してほしいところですが，C にはそのような親切心は見あたりません．

ですから，C で算術演算を行うときは，その結果が正しい範囲にはいるかどうか，プログラミングに際してつねによく気を付けておかなければなりません．面倒なようですが，このソッケなさが C というものなのです（そう達観できれば C プログラマとして一人前）．

ビットト操作を行う —— Cのビット演算記号

コンピュータがデータを2進数の形で扱っているということは，もうご存じの方も多いでしょう．MSX-Cのint型データも，内部的には次のような16ビットの2進数で表されています．

int型データ	2進内部表現
0	0000000000000000
1	0000000000000001
−1	1111111111111111
12345	0110000001110011
−123	1111111110000101

表5.2 int型データの2進内部表現の例

int型のデータを単純な数値データとして取り扱っている限りは，なにもわざわざ2進数を持ち出してくる必要もないのですが，用途によっては2進表現が役にたつ場合もあります．とくにわれわれMSXユーザとしては，グラフィック画面を使ったゲームを作るときに，どうしてもデータをビットパターンとして扱う機能が欲しくなります．

Cの演算記号の中には，数値をこのようなビットパターンの形で操作するものもいくつか用意されています．次の表にその一覧をまとめました．

<table>
<tr><th>演算記号</th><th>意　味</th><th>BASICでの表現</th></tr>
<tr><td>x & y</td><td>ビットごとのANDをとる</td><td>X AND Y</td></tr>
<tr><td>x | y</td><td>ビットごとのORをとる</td><td>X OR Y</td></tr>
<tr><td>x ^ y</td><td>ビットごとのXORをとる</td><td>X XOR Y</td></tr>
<tr><td>~x</td><td>xの全ビットを反転する</td><td>NOT X</td></tr>
<tr><td>x>>y</td><td>xをyビット右にシフトする</td><td>—— なし ——</td></tr>
<tr><td>x<<y</td><td>xをyビット左にシフトする</td><td>—— なし ——</td></tr>
</table>

表5.3 ビット演算子一覧

この表の中で,「&」「|」「^」「~」の4つは,それぞれBASICの「AND」「OR」「XOR」「NOT」とまったく同一の機能を持つ**ビットごとの論理演算子**です.これらは数値データの各ビットごとに,次のような演算を行います.

・「&」演算子――ビットごとのAND(2つのビットが両方とも1なら結果は1)

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

例:xの上位8ビットをすべて0にする

```
x      0011001100110011
y      0000000011111111
       ----------------
x & y  0000000000110011
```

・「|」演算子――ビットごとのOR(2つのビットのうち一方でも1なら結果は1)

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

例:xの上位8ビットをすべて1にする

```
x      0011001100110011
y      1111111100000000
       ----------------
x | y  1111111100110011
```

・「^」演算子――ビットごとのXOR(2つのビットが異なっているとき結果は1)

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

例:xの下位8ビットだけを反転する

```
x      0011001100110011
y      0000000011111111
       ----------------
x^y    0011001100110011
```

・「~」演算子――ビットごとの反転(元のビットが0のとき結果は1)

x	~x
0	1
1	0

例:xの全ビットを反転する

```
x      0011001100110011
       ----------------
~x     1100110011001100
```

図5.16 ビットごとの論理演算

また表の最後に示した2つの演算子「>>」と「<<」は，**ビットシフト演算子**と呼ばれ，データのビットパターンを指定した数だけ右または左にずらす働きがあります(同様な演算子はBASICには存在しません)．この動作を図で表すと次のようになります．

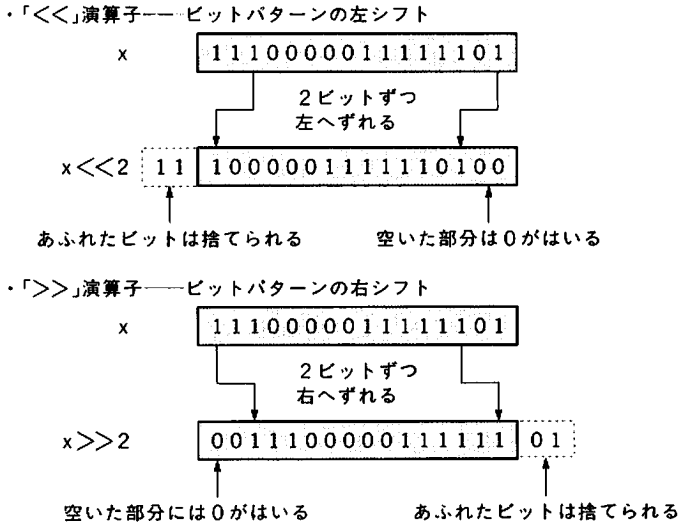



図5.17　ビットシフト演算

文字データに手を加える ── char型変数の操作

int型のデータの操作方法はひととおり説明も終わりました．そこで今度は，char型データに対する操作について述べることにしましょう．といっても実は，char型データのためにとくに用意された演算というものは，Cには1つもありません．

BASICの場合は，文字列データと数値データはまったく別の存在でしたから，文字と数値の混合計算は許されませんでした．次のようなBASICプログラムを実行するとエラーで中断してしまうことは，皆さんご存じのとおりです．

```
100 C$ = "X"
110 C$ = C$ + 1          ←「タイプミスマッチ」エラーが発生
120 PRINT C$
```

ところがCのchar型データは，int型のデータと一緒に計算してもエラーにはなりません．次のプログラムでは，まず変数cに'X'を代入し，その後「c=c+1；」という代入文でcに1が加えられ，最後にそれをputchar()で出力しています．この結果として，画面にはYの文字が表示されるはずです．

```
#include <stdio.h>
main()
{
    char  c;

    c = 'X';
    c = c + 1;
    putchar(c);
}

Y
```

図5.18　char型データとint型データで演算を行う

足し算だけではありません．四則演算でもビット演算でも，int型の数値データに対して行えることは，実はchar型データに対してもまったく同様に実行できるのです．

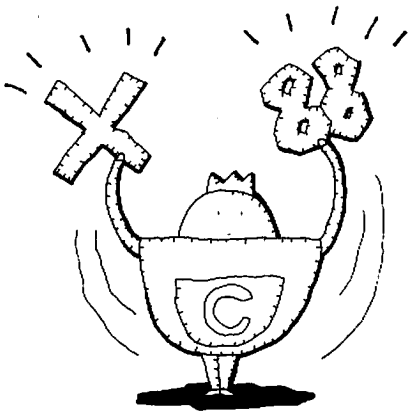
これは，Cコンパイラがchar型のデータを，実際には数値として扱っているからです．たとえば文字定数'X'は，コンパイラにとっては，Xの文字コードに相当する88(16進数では58h)という数値データでしかありません．

この'X'という存在を，文字Xとして見るか，88という数値として見るかは，時と場合とプログラマの判断によります．たとえば，

```
c = 'X';
c = c + 1;
```

を実行するときは，cを数値と考えているわけですし，それを，

```
putchar(c);
```

と表示するときは，今度は変数cの内容を文字と考えているのだといってよいでしょう．Cによるプログラミングは，このあたりも結構イーカゲンです(それがCのよいところでもあるのですけどね)．

5 3 キーボードから入力する

コンピュータの仕事は，せんじ詰めればどれも入力・演算・出力の3要素で成り立っているものです．これは銀行のキャッシュカード端末(金額指定・勘定・支払い)から，家庭用ゲームマシン(スティックを倒す・敵機はいないか・いざ進め)まで，コンピュータと名が付くものならば変わることのない真実です．

本章では，画面出力，データの計算と，ここまでコンピュータの処理の流れを出口のほうから逆にたどってきました．これで最後に入力の方法さえ覚えてしまえば，もう完璧だ！

押されたキーをすぐに読む —— getch()

MSX-Cには1文字入力用のライブラリ関数がいくつか用意されていますが，いちばん使い方がわかりやすいのはgetch()だと思います．次のプログラムにgetch()の使用例を示しました．

```
#include <stdio.h>
main()
{
    char  c;

    c = getch();
    puts("¥nｱﾅﾀｶﾞ ｵｼﾀ ｷｰ ﾊ [");
    putchar(c);
    puts("] ﾃﾞｽﾈ");
}
```

図5.19　1文字入力を行う

このプログラムを実行すると，カーソルを表示して入力待ちになりますから，適当にキーを選んで押してみてください．ここで，たとえばaのキーを押したとすると，すぐに，

```
アナタガ　オシタ　キー　ハ　[a]　デスネ
```

と表示してくるでしょう．getch()によって1文字入力が行われたのです．なお,getch()自身は押したキーを画面にエコーバックしないことに注意してください．

getch()は，ここまで登場した他のライブラリ関数と異なり，**戻り値**に意味がある関数です．関数の戻り値とは，たとえば，

```
c = getch();
```

のように書くと変数cに代入される値です．

BASICでは，INT関数などのように戻り値を返すものを「関数」，PRINT命令のように単に実行されるだけで値を返さないものを「ステートメント」と分けていましたが，Cにはその区別はありません．基本的にCの関数は，どれも戻り値を返します．ただ，その戻り値が意味を持つかどうかの違いがあるだけです．

たとえば，これまで使ってきたputchar()やputs()も戻り値がないわけではありません．これらの関数は，文字の出力に成功すれば0を返し，失敗すると(出力をファイルに向けてリダイレクションしたがディスクが満杯であった場合など)0以外の値を返してそのことを知らせます．ですから，

```
error = putchar();
```

などという代入文を書くことも実はできるのです．ただし，画面にデータを表示する場合は出力に失敗するということがないため，ここまで見てきたようにputchar()やputs()の戻り値は無視してしまってかまいません．

ちょっと話題がずれてしまいました．話をgetch()に戻しましょう．この関数は，キーボードが押されるのを待って，そのキーに相当する文字を返します．BASICのINPUT$関数をご存じの方は，それとちょうど同じ動作をすると思っていただければ間違いありません．

もっともCでは，すでに述べたとおり，文字データとはすなわち文字コードに相当するchar型のデータですから，getch()の戻り値は数値として扱うことが可能です．たとえば次に示すプログラムは，押したキーの次の文字を画面に表示するものですが，ここではgetch()の戻り値に直接1を加えることによって「次の文字」を得ています．

```
#include <stdio.h>
main()
{
    char  c;

    c = getch() + 1;
    puts("¥nｱﾅﾀ ｶﾞ ｵｼﾀ ｷｰ ﾉ ﾂｷﾞﾉ ﾓｼﾞﾊ [");
    putchar(c);
    puts("] ﾃﾞｽﾈ");
}
```

図5.20 getch()の戻り値を数値として扱う

getche()というライブラリ関数は，ほぼgetch()と同様の機能を持っていますが，ただ，入力した文字を画面にエコーバックする点だけが異なります．たとえば次のプログラムでは，1文字入力にgetche()を使っていますから，キーを押すと同時にそれが画面にも表示されます(getch()の動作と比べてみてください)．

```
#include <stdio.h>
main()
{
    char  c;

    c = getche(); ···············getche( )を使用
    puts("¥nｱﾅﾀｶﾞ ｵｼﾀ ｷｰ ﾊ [");
    putchar(c);
    puts("] ﾃﾞｼﾀ");
}
```

図5.21 getche()を使って入力を画面にエコーバックさせる

なお，MSX-CのVer.1.1では，getch()やgetche()は完全にキーボード入力専用の関数となっているため，この関数を使ったプログラムは，実行時に入力リダイレクションを行ってファイルからデータを読み込むことはできません．MSX-C Ver.1.2からはこの制約はなくなり，getch()やgetche()への入力もリダイレクション可能となりました．

入力と出力のカラミ合い —— バッファリングの問題

MSX-C Ver.1.2を使用している方は，getch()による入力を行う場合に気を付けなければいけないことがあります．これは実際にはVer.1.2だけに対する注意点なのですが，Cでプログラミングしていればかならずいつかは出会う問題ですから，Ver.1.1のユーザーの方もぜひ知っておいていただきたいと思います．

この問題とは，画面への文字表示が，文字表示関数の実行と同時には行われないことです．たとえば，次のプログラムを見てください．

```
#include <stdio.h>
main()
{
    char  c;

    puts("INPUT>");
    c = getch();

    puts("¥n[");
    putchar(c);
    puts("] ｶﾞ ｵｻﾚﾏｼﾀ");
}
```

図5.22 Ver.1.1とVer.1.2では動作が異なるプログラム

ざっと眺めたかぎりでは，このプログラムは以下のような動作を行うものに思えます(事実，MSX-C Ver.1.1では，プログラムの動作はここに述べたとおりになります)．

① 「INPUT >」というプロンプトを表示し入力待ちになる
② そこでaのキーを押してみる
③ 「[a]　ガ　オサレマシタ」と表示する

ところが，上のプログラムをMSX-C Ver.1.2でコンパイルして実行してみると，期待通りの結果は得られません．つまり，

① 何も表示されずに入力待ちになる
② しょうがないのでaのキーを押してみる
③ 「INPUT >」と「[a]　ガ　オサレマシタ」の2行が一度に表示され

という動作になってしまうのです．

この理由は，MSX-CのVer.1.2では，画面出力がバッファリングされているからです．バッファリングとは，putchar()などの関数で出力されたデータをすぐに画面に表示せずに，バッファと呼ばれるメモリ領域の中に蓄えておき，バッファがいっぱいになるか，あるいは改行が行われたところで，それをいっきに画面に表示するという方法です．

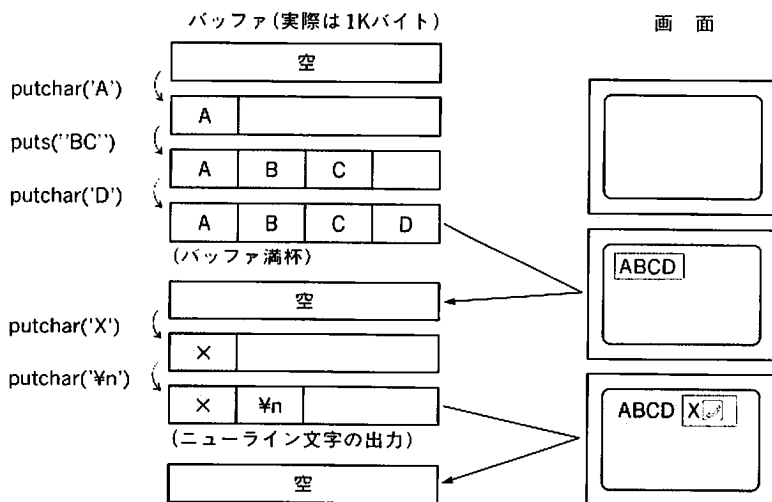



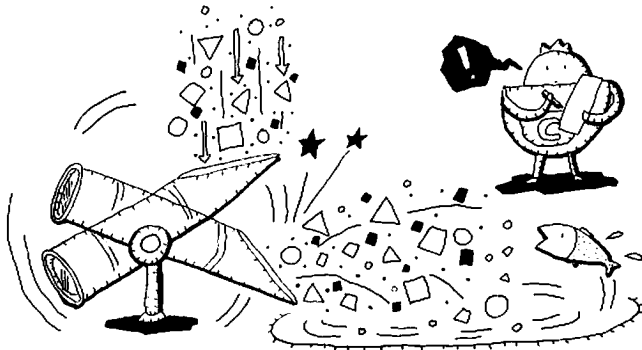
図5.23　バッファリング出力の動作

一般に，バッファリングを行うと，1文字ずつ表示を行うよりも素早い動作が期待できます．しかし上のプログラムで見たとおり，画面表示がバッファリングされていると，あるデータを今すぐ表示して欲しいというときに，それが表示されないという不都合もおこります．これはとくに，画面表示とキーボードからの入力が交互に進行するようなプログラムで問題になるでしょう．

MSX-C Ver.1.2では，このような不都合を避けるために2つの方法が用意されました．1つは画面出力のバッファリングを完全にやめてしまう方法，もう1つは必要な時点で強制的にバッファの内容を画面に吐き出させる方法です．

まずバッファリング機能を停止するには，setbuf()というライブラリ関数を使用します（この関数はVer.1.1には存在しません）．プログラムの最初に1回だけ，setbuf()を次のような形で実行しておくと，画面出力はバッファリングを行わないモードに設定されます．

```
setbuf(stdout,    NULL);
```

ここで，stdoutは「標準出力」，NULLは「ヌルポインタ」を表す単語ですが，これらの意味についてはファイル操作やポインタの説明を行ってから再び詳しく説明します．いまのところは，上のように書けば画面出力のバッファリングが取りやめになるとだけ覚えておいてください．

それでは，このsetbuf()を使って，さきほどのプログラムを書き直してみましょう．

```
#include <stdio.h>
main()
{
    char  c;

    setbuf(stdout, NULL); ·············Ver.1.2ではこの1行を追加する

    puts("INPUT>");
    c = getch();

    puts("¥n[");
    putchar(c);
    puts("] ｶﾞ ｵｻﾚﾏｼﾀ");
}
```

図 5.24 Ver.1.2 で画面出力をバッファリングさせない

こうしてsetbuf()を1行追加するだけで，puts()の出力は関数を実行した瞬間に画面に表示されるようになりますから，前のプログラムのようにプロンプトメッセージがバッファに溜ってしまうことはなくなります．

バッファリングの悪影響を避けるもうひとつの手段は，fflush()を使う方法です(この関数もVer.1.1には存在しません)．fflush()は，その時点でバッファに溜っているすべてのデータを強制的に出力させる働きがあります．この動作は「バッファをフラッシュする」とも呼ばれます．

putchar()やputs()の出力データを画面に表示させるには，fflush()を次の形で使用します．

```
fflush(stdout);
```

前のsetbuf()は，プログラムの最初に1回実行しておけばよかったのですが，このfflush()は，次のプログラムに示すように，画面表示が必要となるごと(要するに入力の直前ごと)に実行しなくてはなりません．

```
#include <stdio.h>
main()
{
    char  c1, c2;

    puts("INPUT first>");
    fflush(stdout); ……………puts( )の出力をフラッシュする
    c1 = getche();

    puts("¥nINPUT second>");
    fflush(stdout); ……………puts( )の出力をフラッシュする
    c2 = getche();

    puts("¥n[");
    putchar(c1);
    puts("] ト [");
    putchar(c2);
    puts("] ｶﾞ ｵｻﾚﾏｼﾀ");
}
```

図 5.25　バッファをフラッシュする

なお，プログラムが終了するときには自動的にすべてのバッファがフラッシュされますから，プログラムの最後に行う画面表示については，fflush()は必要ありません．

文字列を入力する —— gets()

1行ぶんの文字列をまとめて入力したい場合には，gets()というライブラリ関数が用意されています．前に紹介したgetch()は，BASICでいうとINPUT$関数と同じ機能を持っていたわけですが，こちらのgets()はLINEINPUT命令に相当する働きをします．

図 5.26 に，gets()を使った文字列入力の簡単なサンプルを示しました．

このプログラムを実行するとカーソルを表示して入力待ちの状態になりますから，適当な文字列をキーボードから打ち込んで，最後にリターンキーを押してください．

```
#include <stdio.h>
main()
{
    char  s[200];

    gets(s, 200);
    puts("ｲﾝﾌﾟｯﾄ ｻﾚﾀ ﾓｼﾞﾚﾂ ﾊ --- ");
    puts(s);
}
```

図 5.26 文字列入力を行う

たとえば「abc」を入力すると，プログラムは，

インプット　サレタ　モジレツ　ハ　———　abc

と応答を返してくるでしょう．

文字列を入力するときに問題となるのは，C には BASIC のような文字列変数が用意されていないことです．C の char 型の変数は，BASIC の文字列変数と異なり，たった 1 個の文字しか入れておけません．

ではどうしたらよいのか？　その答は配列の利用です．C では char 型の配列変数を使って，文字列を格納したり文字列の操作を行う仕組みになっています．たとえば，上のプログラムの中で，

char　s [200] ;

と書かれた部分は，200 文字ぶんの大きさを持つ char 型の配列変数 s の宣言です．そして，

gets(s,　200);

という gets()の関数呼び出しによって，キーボードから入力した 1 行ぶんのデータが配列 s に代入されます．なお，200 という数値は文字列の長さの最大値を指定するもので，この文字数を超えた入力データは切り捨てられます．

こうして配列変数 s に格納された入力データは，puts()を使って再び画面に表示することもできます．それを行っているのが，プログラムの最後の，

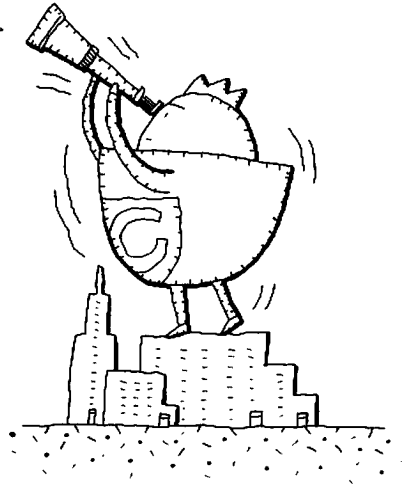
```
puts(s);
```

という関数呼び出しです．

ところで，このプログラムに示したように，gets()やputs()に配列sをパラメータとして与える場合，s[0]あるいはs[200]などとはせず，ブラケット記号を取り去った裸のsという配列名だけを渡しています．これはsという配列全体を指定しているのです．

BASICの配列は，S(0)やS(200)という個々の要素として利用はできても，全体を一括して取り扱うことはできませんでした．それに対してCでは，上の例のように1個の配列をまるごと操作の対象として扱えるようになっています．この機能を利用すると，いろいろと面白いプログラミングテクニックも考えられるのですが，その実例は章をあらためて紹介することにしましょう．

6章

Cのプログラムは如何にして組み立てられるのか

プログラムの大事な要素の1つに、処理の流れを変える機能があります．ある決まった動作を上から順に実行するだけでは，どんなプログラムもたいしたことはできません．状況に応じていろいろと対応を変えられなくてはプログラムなんて意味がない！　とさえいえるかもしれません．

ところで一口に「処理の流れ」といっても，細かく見れば，世の中に存在するプログラムの数と同じだけの処理の流れがあるわけですが，それらは結局のところ，条件によって動作を選択する「条件判断」か，ある動作を繰り返す「繰り返し」の2種類に大きく分けられます．

以下この章では，その2つの処理をCのプログラムで実現する方法について説明していきます．

6 1 条件判断

ファジー(あいまい)論理とやらがトレンディな昨今ですが，白か黒か2つに1つの答を選択する条件判断は，やはりプログラミングには欠かせない要素です．イエスかノーかの単純な選択といってもバカにはできません．イエスノーの質問も，20回繰り返せば森羅万象が弁別できるのです(しかし「二十の扉」などという太古の番組は，今や誰も知らないか)．

プログラムの最小要素 —— 文

条件判断の説明にはいる前に，プログラムの流れの最小単位である「文」について，きちんと押さえておきましょう．文，などと改まって書くとどうも小難しくなっていけませんが，要するに「ある処理を実行するもの」をCではすべて文と呼んでいます．

ここまでの説明で出てきた文には，次の2種類がありました．

・代入文　(変数にデータを代入する)

```
i = 1234 ;
c = 'a' ;
c = getch() ;
```

・関数呼び出し文　(関数を単独で呼び出し，処理を行う)

```
putchar('a') ;
puts("abc") ;
```

これらの文は，最後がセミコロン「；」で終わっているのが大きな特徴です．セミコロンは，代入文や関数呼び出し文の一部であって，これを含めて文が成り立っているものと考えてください．

さて，こうして今まで出てきた文を2種類にまとめてみると，実はこれでCのプログラムの本質はすべて尽くされてしまいました．Cのプログラムとは，要はこの2つの単純な文を巧妙に組み立てたものにすぎません．その組み立てを行うものが，これから述べる「条件判断」と「繰り返し」なのです．

条件の判定 —— if文

Cの条件判断の基本となるのはif文です．Cのif文の書式は次のとおりです．

if (条件)　文

このif文の意味は，「条件」が成立すれば「文」を実行せよ，ということです．BASICのIF文とよく似ていますが，Cのif文では条件の後ろにTHENに相当するキーワードを書く必要はありません(書いてはいけません)．また条件はかならずカッコに入れて書かなくてはなりません．

図6.1に示すのはif文を使ったプログラムの例です．このif文は，s以上のキーが押されれば，「ピンポーン」と表示します．

```
#include <stdio.h>
main()
{
    char  c;

    c = getche();

    if (c >= 's')
        puts("¥nﾋﾟﾝﾎﾟｰﾝ");
}
```

図6.1　if文による条件判断

このプログラムに示したように，Cのif文はBASICのIF文と違い全体を1行に詰め込む必要はありません．Cのプログラムはフリーフォーマット形式と呼ばれ，プログラム中の単語の並べ方が，すべてプログラムを書く人間

のセンスにまかされていることは前にも述べたとおりです．

ただし一般には，次のような書き方がよく用いられます．本書でもこの形式を採用することにしました．

```
if (c >= 's')          ←if から条件までを 1 行に書く
    puts("¥n ピンポーン");
      ↑
    実行文は 1 段(本書では 4 文字)下げて次行に書く
```

さて，条件が成立しなかった場合にもなんらかの動作をさせたい場合には，else 付きの if 文を使用します．else 付きの if 文の書式は次のとおりです．

```
if (条件)　文 1　else 文 2
```

この else 付きの if 文は，「条件」が成立したときに「文 1」，成立しなかったときは「文 2」を実行します．else に釣られて BASIC の IF 文のクセが出て，くれぐれも if の後ろに then などと書かないように(コンパイラに怒られます)．

図 6.2 は else 付きの if 文の実例です．else 付きの if 文は，このように if と else の頭をそろえて書くとプログラムが見やすくなります．

```
#include <stdio.h>
main()
{
    char  c;

    c = getche();

    if (c >= 's')
        puts("¥nﾋﾟﾝﾎﾟｰﾝ");
    else
        puts("¥nﾌﾞｰｰｰｯ");
}
```

図 6.2　else 付きの if 文による条件判断

if文は，それ自体で1つの文とみなされます．ですから，if文の実行文として，さらに別のif文を書くこともできます(このような状態をif文の「入れ子」と呼びます)．

たとえば図6.3のプログラムを見てください．これは押されたキーが数字の1から5までの範囲ならば「ヨシ」と表示し，そうでなければなにもしないプログラムですが，いま述べたif文の入れ子を利用しています．

```
#include <stdio.h>
main()
{
    char  c;

    c = getche();

    if (c >= '1')
        if (c <= '5')
            puts("¥nﾖｼ");
}
```

図6.3　if文を組み合わせる「入れ子」

また，こういったif文の入れ子構造の中にelseがはいる場合，Cではそのelseはかならず直前の(ただしelseを持っていない)ifにつながるものと決められています．たとえば図6.4のようなプログラムでは，elseは2番目のifに対応するものとみなされます．

```
if (c >= '3')
    if (c <= '6')
        puts("¥n3 ﾄ 6 ﾉ ｱｲﾀﾞ ﾃﾞｽ");
    else
        puts("¥n7 ｲｼﾞｮｳ ﾃﾞｽ");
```

こちらは無関係

このifとelseが対応する

図6.4　if文の入れ子とelseの対応

文をまとめる —— 複文

if 文の条件が成立したとき，あるいはしなかったときに，複数の動作を行わせたければ，それらの実行文を「ブロック」にまとめます．ブロックとは，実行文を次のようにブレース記号「{」と「}」で囲んだものです．また，このように複数の文をブロックにまとめたものを「複文」とも呼びます．なおブロックの後ろにはセミコロンを付ける必要がありません．

{ 文1 文2 文3 …… }

↑ここにはセミコロンは必要ない

次のプログラムに複文の使用例を示します，ここでは if 文の条件が成立したとき，複文を使って 3 つの関数呼び出しを行わせています．

```
#include <stdio.h>
main()
{
    char   c;

    setbuf(stdout, NULL); ……………MSX-C Ver.1.2にのみ必要
    puts("[s] ｲｼﾞｮｳ ﾉ ｷｰ ｦ ｵｼﾃ ｸﾀﾞｻｲ : ");
    c = getche();

    if (c >= 's')
        { puts("¥nﾅｰｲｽ! "); putchar(c); puts(" ﾊ s ｲｼﾞｮｳﾃﾞｽﾈ"); }
}
                3つの関数呼び出しをまとめた複文
```

図 6.5 複文の使用例

ところで上のプログラムでは，複文というものが 1 個の文であることを示すため，全体をわざと 1 行に並べて書きました．しかしこのような書き方はあまり見やすいものではありません(でしょ？)．そこでエレガントなプログラミングを志すわれわれとしては，これをなんとか見やすく美しいプログラムにしようと努力するわけです．

複文の見やすい書き方には多くの流派があり，プログラマが10人いれば1ダースの書き方があるといわれるくらい，その表記法はさまざまですが，本書では原則として図6.6の書式を採用することにしました．

```
if (c >= 's') {
    puts("¥nﾅｰｲｽ! ");
    putchar(c);
    puts(" ﾊ s ｲｼﾞｮｳ ﾃﾞｽﾈ");
}
```

図6.6 本書で採用するif文の書式

また，この後ろにさらにelseが続く場合には，図6.7のような位置にelseを置きます．

```
if (c >= 's') {
    puts("¥nﾅｰｲｽ! ");
    putchar(c);
    puts(" ﾊ s ｲｼﾞｮｳ ﾃﾞｽﾈ");
}
else { ··············本書ではこの位置にelseを書く
    puts("¥nﾁｶﾞｳﾃﾞｼｮ! ");
    putchar(c);
    puts(" ﾊ s ﾖﾘ ﾁｲｻｲ ﾖ");
}
```

図6.7 本書で採用するelse文の書式

データの大小を比較する —— 比較演算子

前項のプログラムには，データの大小を比較する「＞＝」という比較演算記号が登場しました．これはBASICの場合と同様に「大きいか等しい」という意味を持っています．

CとBASICの記号の使い方がこの調子でまったく同じならば，皆さんも筆者も幸福だったのですが，実際にはCで使う比較演算記号はBASICのそれと少々異なる部分があります．そこで，条件判断の説明を先に進める前に，Cの比較演算記号をまとめて紹介しておくことにしました．表6.1がCの比較演算記号の一覧です．

比較演算	意　味	BASICでの表現
x＞y	xはyより大きい	X＞Y
x＞＝y	xはyより大きいか等しい	X＞＝Y(またはX＝＞Y)
x＜y	xはyより小さい	X＜Y
x＜＝y	xはyより小さいか等しい	X＜＝Y(またはX＝＜Y)
x＝＝y	xはyと等しい	X＝Y
x！＝y	xはyと等しくない	X＜＞Y(またはX＞＜Y)

表6.1　Cの比較演算記号一覧

この中でとくに注意が必要なのは，2つの値が等しいことを表す「＝＝」という記号です．

BASICでは，Xの値がYに等しいことを表すにも「X＝Y」，XにYを代入する場合も「X＝Y」と書きましたが，Cでは両者をきちんと区別しなくてはなりません．上の表に示したように，Cのプログラムでは，xとyが等しいということは，

```
x == y          (xとyは等しい)
```

とイコール記号を2つ重ねて書きます．

そしてxにyを代入するときだけ，

x = y　　　　　(xにyを代入)

とBASICと同じ代入記号(イコール記号1個)を使うのです．

この2つの記号の使い分けは非常に重要です．というのは，Cでは比較記号「==」と間違って代入記号「=」を使ってもエラーにはならず，それなりの処理が実行されてしまうからです．たとえば図6.8のプログラムを見てください．

```
#include <stdio.h>
main()
{
    char  c;

    c = getche();
    if (c = 'y') ……………ここに間違いがある
        puts("イエス");
    else
        puts("ノー");
}

┌────実行結果
↓
n    ……………nを入力したが...
イエス        イエスと表示されてしまった
```

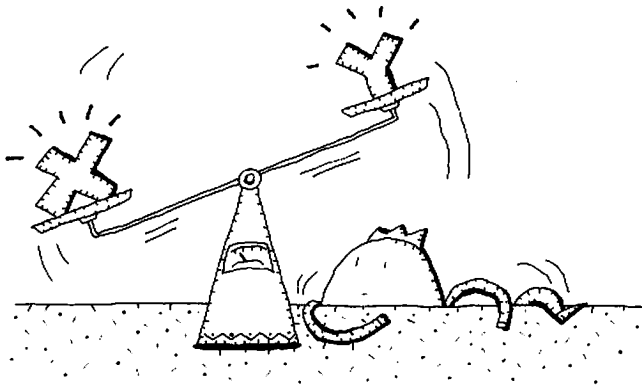
図6.8　比較演算と代入を間違えたプログラム

これは，キーボードからyが入力されればイエス，それ以外の文字の場合はノーと表示するプログラムを作ったつもりなのですが，実行してみると，どのキーを押してもイエスの答しか返ってきません．

その原因はif文の中の「==」と「=」の書き間違いです．つまりイコール記号が1個足りないばかりに，「変数cが'y'に等しければ」という条件ではなく，「変数cに代入した'y'という値が真ならば」という意味に化けてしまったのです(後で述べるようにCの条件判断では0以外の値はすべて真とみなされますから，この場合if文の条件は常に成立してしまうことになります)．

慣れないうちはどうしてもBASICのクセが出て，「==」を使用するべき

場所で，つい代入演算子「＝」を使ってしまうことも多いと思いますが，くれぐれも注意を怠らないようにしてください．

また，Cではxとyが等しくないことを「x!＝y」と表します(!と＝を重ねれば≠になる，と考えると覚えやすいでしょう)．大小比較の記号についてはBASICと一緒です．ただし，BASICでは「より大きいか等しい」ということを表すのに「＝＞」という記号が使えましたが，Cには「＞＝」のほうしか用意されていません．同様に「より小さいか等しい」を表す記号も「＜＝」という書き方しか許されません．

6-2 複雑な条件判断

Cのif文は，初めのうちはBASICのIF文と比べると，どうも難しく感じられるようです．とくに人の作ったプログラムなどを見ると，if文の条件に尋常ではない式が書かれていて思わず頭をかかえることが多い．そんな場合は大抵，ここで紹介する論理値，論理演算子，代入式のどれかがカラんでいるはずです．というわけで，ここではその3つのアイテムの説明を行います．

条件判断のしくみ —— 論理値

ここまでは，if文の条件として，いわゆる「条件式」だけを使ってきましたが，実はif文のカッコの中にはどんな計算式を書いてもかまいません．たとえば次のようなif文も，立派なCのプログラムなのです．

```
if (error)
    puts("ガチョーン");
```

このような場合，条件が成り立つかどうかは，カッコ中の式の値によって以下のとおりに判定されます．

式の計算結果が0以外 —— 条件は成立
式の計算結果が0　　 —— 条件は不成立

ですから上のif文は，変数errorの値が0でなければ「ガチョーン」と表示するという意味になります(0以外ならば，1でも2でも−1でも結果に変わりありません)．

このように，ある数値データが0かそれ以外か，つまりif文の条件を成立させるかどうかという点だけに注目することを，「データを論理値として扱う」といいます．そしてデータを論理値として扱うとき，0以外の値を「真」，

0を「偽」と呼びます(表6.2).

論理値	その実態
真	0以外の数値
偽	0

表6.2　論理値とその値

Cのif文は，原則としてこの論理値を仲立ちとして実行されるしくみになっています．これは通常の条件演算の場合も例外ではありません．たとえば，

```
if (c <= '1')
    puts("チイサスギマス");
```

というif文は，まずcと'1'を比較して真または偽の値を得て，それをif文が判断する2段構えの処理になっているわけです．

なお，比較演算の結果としての「真」の値は1と決っています．つまり比較演算の答は，かならず0または1になるのです．これは次のプログラムで確かめることができます．

```
printf("%d", (2 > 1));    ←　真なので1が表示される
printf("%d", (2 < 1));    ←　偽なので0が表示される
```

条件を組み合わせる —— 論理演算子

複雑な条件判断を行うには，ifとelseを組み合わせて使うのもひとつの方法ですが，もうひとつ，「AかつB」や「AまたはB」などという，いわゆる論理演算を利用するという手もあります．

BASICの場合は論理演算にはビット演算と同じ記号(AND, ORなど)を使いました．それに対してCでは，すでに紹介した「&」や「|」などのビット演算子とは別に，論理演算専用の演算子が用意されています．その一覧を表6.3に示します．

演算記号	意　味
X && y	x が真で，かつ y も真(論理AND)
x | | y	x が真か，または y が真(論理OR)
! x	x ではない(論理NOT)

注：「論理XOR」に相当する演算子はありません．

表 6.3　論理演算子一覧

● 「&&」演算子 —— それに加えて〜なら(論理 AND)

「x && y」は，x という条件が成立し，かつ y という条件も成立することを表します(「&」と「&」の間には空白を入れてはいけません)．

次に示すプログラムは，c の値が'a'以上でかつ'z'以下，つまりアルファベットの小文字の範囲にはいっていれば「OK」と表示します．

```
#include <stdio.h>
main()
{
    char  c;

    setbuf(stdout, NULL); ……………MSX-C Ver.1.2にのみ必要
    puts("a ｶﾗ z ﾏﾃﾞﾉ ｷｰ ｦ ｵｼﾃ ｸﾀﾞｻｲ : ");
    c = getche();

    if (c >= 'a' && c <= 'z')
        puts("¥nOK");
}
```

図 6.9　「&&」演算子の使用例

● 「| |」演算子 —— または〜なら(論理 OR)

「x | | y」は，x が成立しているか，または y が成立しているということを表します(これもやはり「|」と「|」の間に空白を入れてはいけません)．

次に示すのは，「| |」演算子を使って，y か Y のキーが押されたときに「OK」と表示するプログラムです．

```
#include <stdio.h>
main()
{
    char  c;

    setbuf(stdout, NULL); ···············MSX-C Ver.1.2にのみ必要
    puts("ﾖﾛｼｲ ﾃﾞｽｶ (y/n) ? ");
    c = getche();

    if (c == 'y' || c == 'Y') ···············「yまたはYならば」という条件を表す
        puts("¥nOK");
}
```

図6.10 「||」演算子の使用例

● 「!」演算子 ── ～でない場合は(論理NOT)

「!x」は，xが成立しなければ，という条件を表します．次のプログラムは，押されたキーがnでもNでもなければ「OK」と表示します．

```
#include <stdio.h>
main()
{
    char  c;

    setbuf(stdout, NULL); ···············MSX-C Ver.1.2にのみ必要
    puts("ﾖﾛｼｲ ﾃﾞｽｶ (y/n) ? ");
    c = getche();

    if (!(c == 'n' || c == 'N')) ···············「(nまたはN)でなければ」
        puts("¥nOK");                            という条件を表す
}
```

図6.11 「!」演算子の使用例

代入した値を判定する —— 代入式

次にここで，Cの条件判断の特徴である「代入式」の利用法について説明しておきましょう．これは効率的なプログラミングの道具でもあり，またプログラムをわかりにくくする元凶でもあるというCに特有のユニークな存在です．

代入式とは簡単にいえば，代入文から最後のセミコロンを取り除いたものです．つまり，

```
x = 1;
```

という代入文からセミコロンを取り除けば，

```
x = 1
```

という代入式になるわけです．代入式は変数に値を代入した後，その値を代入式自身の値として利用できます．これが最もよく利用されるのは，図6.12のプログラムに示すように，変数に代入した値がある条件を満たすかどうか，すぐに調べる場合です．

```
#include <stdio.h>
main()
{
    char  c;

    if ((c = getche()) == 'y')
        puts("OK");
}
```

図6.12　変数に代入した値をそのまま利用する

このプログラムのif文は，まず変数cにgetche()で1文字入力し，その入力した文字がyであれば「OK」と表示します．

6-3 繰り返し

以上でif文を使った条件判断の説明は終わりにして，ここから繰り返し処理のプログラミングについて述べることにします．

回数を指定した繰り返し —— for文

Cのプログラムでループを作る方法には，いくつか種類がありますが，一番なじみやすいのは，BASICのFOR～NEXT命令とよく似たfor文でしょう．Cのfor文の書式は次のとおりです．

for (初期設定 ; 実行条件 ; 再設定) 文

このfor文を実行すると，最初に「初期設定」を1回だけ行い，そして「実行条件」が成立している間「文」を実行しては「再設定」を繰り返します．この動作をフローチャートで表せば，図6.13のようになります．

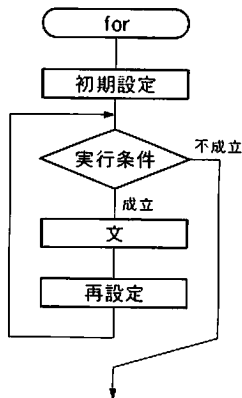



図6.13　for文のフローチャート

図6.14のプログラムにfor文の実例を示します．これは，i=1から始めて，i<=10である間，iを1ずつ増やしながらループするわけですから，結局文字Aが10個表示されることになります．

```
#include <stdio.h>
main()
{
    int  i;

    for (i = 1; i <= 10; i = i + 1)
        putchar('A');
}


AAAAAAAAAA  ←――― 実行結果
```

図6.14　for文を使ってputchar()を10回実行する

このように，ある変数を初期値から終値まで一定の値を加えながら繰り返すという動作は，for文の最も基本的な使い方といってよいでしょう．これは，BASICのFOR～NEXT命令とまったく同じ動作でもあります．実際に，

FOR I＝初期値　TO 終値　STEP 増減値　……

というBASICのFOR～NEXT命令は，次のfor文に機械的に置き換えることができます(変数の値が増加するか減少するかによって，条件判定の不等号の向きが異なるところに注意してください)．

for (i＝初期値；　i＜＝終値；　i＝i＋増加値)　……
for (i＝初期値；　i＞＝終値；　i＝i-減少値)　……

しかしCのfor文は，このような単純な用途以外に，もっといろいろな使い方ができるところに大きな特徴があります．たとえば，for文のループ変数は，値を足していけるだけではありません．図6.15のプログラムは，iの値を1から始めて2, 4, 8, ……と2倍しながら256まで増加させることを意味しています．

```
#include <stdio.h>
main()
{
    int  i;

    for (i = 1; i <= 256; i = i * 2) {
        printf("%d", i);
        putchar(' ');
    }
}


1 2 4 8 16 32 64 128 256  ←―― 実行結果
```

図 6.15　for 文のループ変数を倍加させる

また次のプログラムは，まず c に'n'を代入しておいて，c が'y'に等しくなるまで 1 文字入力を続けることを意味します．これなどは C の for 文としては正しいものですが，もう BASIC の FOR～NEXT 命令からの類推では動作を理解できないかもしれませんね．

```
#include <stdio.h>
main()
{
    char  c;

    for (c = 'n';  c != 'y';  c = getche())
        puts("¥nOK ﾃﾞｽｶ (y/n) ?¥n");

    puts("ｿﾚﾊ ﾖｶｯﾀ");
}
          ↓―――――― 実行結果
OK ﾃﾞｽｶ (y/n) ?
n
OK ﾃﾞｽｶ (y/n) ?
N
OK ﾃﾞｽｶ (y/n) ?
y
ｿﾚﾊ ﾖｶｯﾀ
```

図 6.16　自由度の高い for 文の書式

要するに，C の for 文では「初期設定」と「実行条件」と「再設定」の組み合わせいかんで，どんなループ条件でも表せるようになっているのです．はじめのうちは，あまり妙な条件を設定するのも考えものですが，for 文は非常に便利な使い方ができるということだけは覚えておいてください．

空ループによる時間かせぎ —— 空文

BASIC では，単なる時間かせぎのために，

FOR I=1　TO 1000：　NEXT

という何もしないループ(空ループ)を使うことがありました．C の for 文で同様なことを行うには，

```
for (i=1;  i<=1000;  i=i+1)
        ;
```

と書きます．この for 文は，繰り返しの本体に，ただひとつのセミコロンが置かれています．C ではこうやって何も実行しない文を表現します．このセミコロンだけの文を，とくに「空文(くうぶん)」と呼びます．

図 6.17 のプログラムに空文の利用例を示しました．ここでは空文を 6000 回繰り返していますが，普通の MSX マシンならば，これで約 0.2 秒の時間待ちとなります．

```
#include "stdio.h"
main()
{
    int  i, t;

    setbuf(stdout, NULL); ··············MSX-C Ver.1.2にのみ必要

    for (i = 1; i <= 1000; i = i + 1) {
        putchar('A');
        for (t = 1; t <= 6000; t = t + 1)   ─┐ 6000回の空ループ
            ;                              ─┘ (0.2秒)
    }
}
```

図 6.17　for 文を使った空ループ

省エネ方式の代入文 —— 代入演算子

ここまで，for 文のループ変数を再設定する部分は，「i=i+1」や「i=i＊2」などのように，ごく普通の書き方を使ってきましたが，C ではこういった変数の値の更新には表 6.4 のような「代入演算子」を利用することもできます．というより，こう書くほうが C の世界では一般的です．

演算子	意 味
x ＋= 数値	x ＝ x ＋ 数値
x －= 数値	x ＝ x － 数値
x ＊= 数値	x ＝ x ＊ 数値
x ／= 数値	x ＝ x ／ 数値

表 6.4　代入演算子一覧

たとえば，変数 x の値を 5 だけ増加させるなら「x＋＝5」，x の値を 3 倍にしたければ「x＊=3」という表現になるわけです．

さらに，プログラムの中でとくに使用する機会が多い「変数に 1 を加える操作」と「変数から 1 を減ずる操作」は，表 6.5 のようにもっと簡単に表現することもできます．この「＋＋」をインクリメント演算子，「－－」をデクリメント演算子と呼びます．

演算子	意 味
＋＋x	x ＝ x ＋ 1
－－x	x ＝ x － 1

表 6.5　インクリメント演算子とデクリメント演算子

これらの演算子を使うと，for 文は図 6.18 のように，少しだけ簡潔に表現できることになります．

```
#include <stdio.h>
main()
{
    int  i;

    for (i = 1; i <= 10; ++i) { ·············· i を1から10まで1ずつ増やしていく
        printf("%d", i);
        putchar(' ');
    }
    putchar('¥n');

    for (i = 1; i <= 2187; i *= 3) { ······ i を1から2187まで3倍ずつ増やしていく
        printf("%d", i);
        putchar(' ');
    }
    putchar('¥n');
}
```

図 6.18 for 文の条件を簡潔に書く

なお，代入演算子やインクリメント／デクリメント演算子は，for 文専用というわけではなく，プログラムのどこで使ってもかまいません．そしてまた，これらの演算子を工夫して使うと，非常に効率のよいプログラムを書くことができます．それらの実例もおいおい紹介していきましょう．

条件が成立するまで繰り返す —— while 文と do 文

さて，ここまで紹介してきた for 文は，原則としてループ変数でコントロールされる動作を扱うものでした．しかし，通常のプログラミングで出てくるループは，特定のキーが押されるまで繰り返すとか，ファイルを最後まで読み込むとか，ループ変数を持たないものも少なくありません．そのような繰り返し処理のために用意されたものが，while 文と do 文です．

まず，while 文の書式を一般的な形で示すと次のようになります．

while（条件） 文

これは，「条件」が成立している間，「文」を繰り返すという意味になります．図 6.19 は while 文の動作をフローチャートで表したものです．

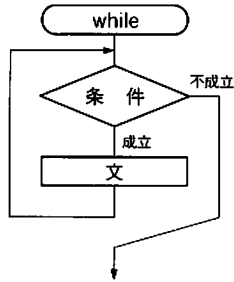



図 6.19　while 文の動作を表すフローチャート

次のプログラムに while 文の実例を示します．このプログラムは，'q'のキーが押されるまで入力した文字を画面に表示し続けます．

```
#include <stdio.h>
main()
{
    char  c;

    setbuf(stdout, NULL); ··············MSX-C Ver.1.2 にのみ必要

    while ((c = getch()) != 'q')
        putchar(c);
}
```

図 6.20　while 文を使ったループ

while 文はループの最初で条件の判定を行うものですが，場合によっては，何かある動作を行った結果でループを続行するかどうかを決めたほうが都合のよいこともあります．そのために用意されたものが do 文です．

do 文の書式は次に示すとおりです．

do 文　while (条件)；

この do 文は，まず「文」を実行してから「条件」を判断し，それが成立していればループを繰り返します．図 6.21 はこの動作をフローチャートで表したものです．

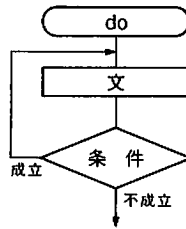



図6.21 do文のフローチャート

なおdo文の書式は本来は上に示したとおりなのですが，実行する文が1個だけの場合，プログラムは，

```
do
        文
while (条件) ;
```

という形で書かれることになり，「while」をwhile文の始まりと見誤りやすくなって好ましくありません．

そこで一般には，do文は実行文が1個だけでもかならずブロックに閉じ込めて，

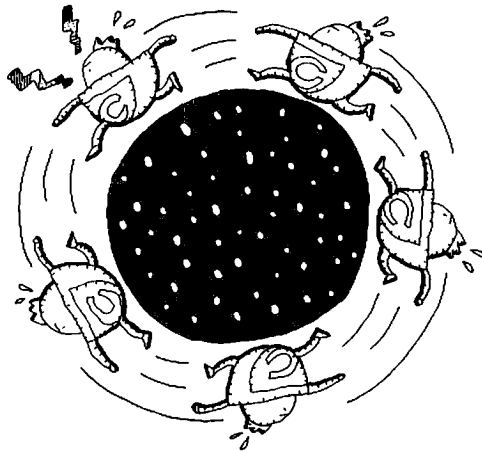
```
do  {
          文
}   while (条件) ;
```

という書き方をします．つまりdo文はかならず「do{」で始まり，「}while」で終わるものと考えてしまうわけです．本書でもこの方法をとることにしました．

図6.22に示すプログラムは，do文を使った実例です．ここでは，数字の1から5までのどれかのキーが押されるまで1文字入力を繰り返しています．

```
#include <stdio.h>
main()
{
    char  c;

    do {
        puts("1 ｶﾗ 5 ﾏﾃﾞﾉ ｽｳｼﾞ ｦ ﾆｭｳﾘｮｸ ｼﾃ ｸﾀﾞｻｲ¥n");
        c = getche();
        putchar('¥n');
    } while (c < '1' || c > '5');
}
```

図 6.22　まず「文」を実行してから条件判断をする do 文

6-4 それ以外の制御構造

ここまで紹介した制御文のほかに，Cにはループ文の補助的な存在である break 文と continue 文，そして場合わけをするのに便利な switch 文の3つが用意されています．

ループから抜け出す —— break 文

break 文は，次のような書式を持っています．

```
break ;
```

break 文は基本的にループの中で実行してはじめて意味を持つ文です．この文を実行すると，ループを強制的に中断して途中から抜け出すことができます(図 6.23)．

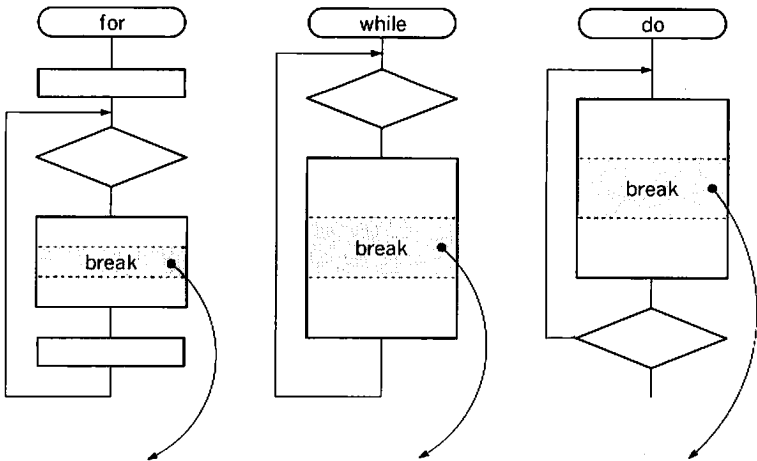



図 6.23 break 文の動作

図 6.24 に示すプログラムは，break 文の動作を示す簡単なサンプルです．このプログラムは基本的には for 文を使って 5 個の文字を入力するものですが，入力された文字が q であった場合，break 文を実行してループからの途中脱出を行います．

```
#include <stdio.h>
main()
{
    int   i;
    char  c;

    for (i = 1; i <= 5; ++i) {
        printf("%d", i);  puts("ﾊﾞﾝｺﾞｳ ﾉ ﾓｼﾞ ﾊ ?");
        c = getch();
        if (c == 'q') {
            puts(" BREAK!!¥n");
            break;
        }
        puts(" ["); putchar(c); puts("]¥n");
    }
}
                    ┌────────────実行結果
                    ↓
1ﾊﾞﾝｺﾞｳ ﾉ ﾓｼﾞ ﾊ ? [a]        ……………a を押した
2ﾊﾞﾝｺﾞｳ ﾉ ﾓｼﾞ ﾊ ? [b]        ……………b を押した
3ﾊﾞﾝｺﾞｳ ﾉ ﾓｼﾞ ﾊ ? BREAK!     ……………q を押すと強制終了
```

図 6.24　break 文の使用例

ループの処理を1回だけスキップする —— continue文

break 文と少し似ているものに，continue 文があります．書式は以下のとおりです．

```
continue ;
```

continue 文もループの中で使ってはじめて意味を持つ文ですが，こちらはループから完全に抜け出してしまうのではなく，その回のループ本体の実行を 1 回分だけキャンセルする働きがあります．つまり，continue 文があると処理はそこで止まり，次の繰り返しに進むのです．この動作は図 6.25 のように表せます．

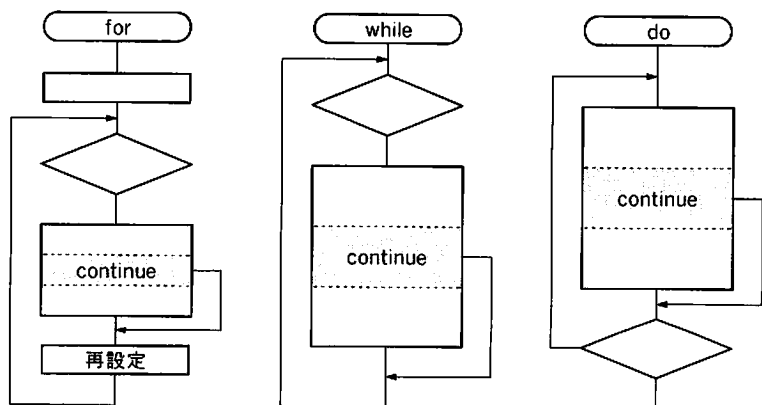



図 6.25 continue 文の動作（各ループ文に対する 3 種）

図 6.26 に示すプログラムは，さきほどのプログラムの break 文を continue 文に書き換えたものですが，こちらは q という文字を入力してもループは中断しません．ただループの中で continue 文以降にある部分がスキップされるだけとなっています．

```
#include <stdio.h>
main()
{
    int   i;
    char  c;

    for (i = 1; i <= 5; ++i) {
        printf("%d", i);  puts("ﾊﾞﾝﾒ ﾉ ﾓｼﾞ ﾊ ?");
        c = getch();
        if (c == 'q') {
            puts(" BREAK!¥n");
            continue;
        }
        puts(" ["); putchar(c); puts("]¥n");
    }
}
```

実行結果

```
1ﾊﾞﾝﾒ ﾉ ﾓｼﾞ ﾊ ? [a]       ……………a を押した
2ﾊﾞﾝﾒ ﾉ ﾓｼﾞ ﾊ ? [x]       ……………x を押した
3ﾊﾞﾝﾒ ﾉ ﾓｼﾞ ﾊ ? BREAK!    ……………q を押してもループは終了しない
4ﾊﾞﾝﾒ ﾉ ﾓｼﾞ ﾊ ? [s]
5ﾊﾞﾝﾒ ﾉ ﾓｼﾞ ﾊ ? [k]
```

図 6.26 continue 文の使用例

ループの奥底からの脱出 —— goto 文

何重にも入れ子になったループの中で break 文や continue 文を実行する場合，これらの文が効力を持つのは，その時点で実行している一番内側のループだけに限られます．

たとえば図 6.27 のような 2 重ループのプログラムでは，break 文で抜け出せるのは変数 j を使った内側のループだけで，いっきに一番外まで脱出することは break 文では不可能です．

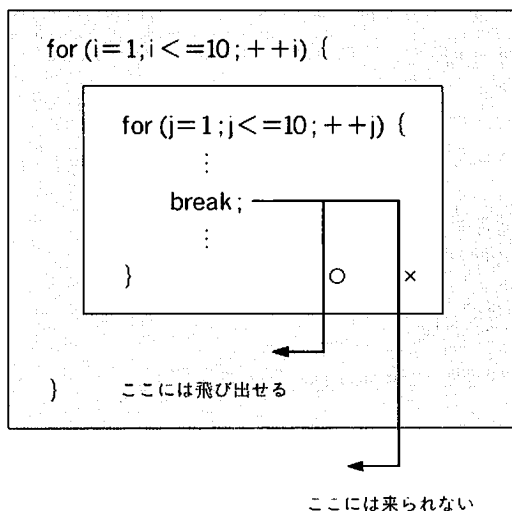



図6.27　2重ループからの脱出

そこで用意されたプログラム制御文が goto 文です．goto 文は適当なラベル付きの文に，無条件でプログラムの流れを移すことができます．ラベル付きの文と goto 文の書式は，それぞれ以下のとおりです．

```
ラベル名：　文
```

ラベル名は goto 文の飛び先を示す目印で，変数名と同様にアルファベットの大文字と小文字，およびアンダースコア記号の組み合わせであれば，自由に名付けてかまいません．

なんらかの文の前にラベル名とコロン記号「：」を置いたラベル付きの文は，goto 文の飛び先として指定できます．なお，これはあくまでも「ラベル付きの文」でありますから，ラベルの後にはかならず文(空文でもよい)が必要です．

```
goto ラベル名；
```

そして goto 文は，指定されたラベル付きの文にジャンプします．これは飛び先がどこにあってもかまいません(注：ただし同じ関数の中であること)．

ですから多重の入れ子になったループの内側から，一番外側まで抜け出すことも簡単です．

図6.28に2重のループの中からgoto文によっていっきに脱出するプログラムの例を示します．なお，ここで使っているkbhit()は，その時点でキーが押されていれば真，押されていなければ偽の論理値を返すライブラリ関数です．

```
#include <stdio.h>
main()
{
    int  i, j;

    for (i = 1; i <= 100; ++i) {
        for (j = 1; j <= 100; ++j) {
            puts("¥ni + j = ");
            printf("%d", i + j);
            if (kbhit()) {
                puts("¥nBREAK!¥n");
                goto  exit_all;
            }
        }
    }
exit_all:
    puts("end");
}
```

図6.28　goto文の使用例

効果的な場合わけ処理 —— switch文

さて最後に残ったのは，Cの制御構造の中でも最もクセが強くて扱いにくく，それだけにイザというとき頼りになるswitch文です．これはBASICでいうならばON～GOTO命令に相当する「多重分岐」のための制御文です．switch文の書式は次のとおりです．

```
switch (式)  {
    case 定数 1：  文 1-1  文 1-2  ……
    case 定数 2：  文 2-1  文 2-2  ……
                  ⋮
    default：     文 n-1  文 n-2  ……
}
```

switch 文は，まず「式」の値を計算します．そしてその値と「case 定数：」の形で書かれた定数値を上から比較していき，両者が等しくなった部分以降の文をすべて実行します．式の値がどの定数とも一致しなければ，「default：」以降の文を実行します．

この動作は，あれこれ説明するよりも実例を見ていただくほうがわかりやすいかもしれません．図 6.29 に示すのは，switch 文の動作を見るための簡単なプログラムです．

```
#include <stdio.h>
main()
{
    switch (getche()) {
     case 'a':
        puts("¥nApple");
     case 'b':
        puts("¥nBanana");
     default:
        puts("¥nNot found");
    }
}

  ┌──実行結果 1
a ↓       ……………… a を入力した
Apple     ………………「case'a'：」以降の文がすべて実行される
Banana
Not found


  ┌──実行結果 2
b ↓       ……………… b を入力した
Banana    ………………「case'b'：」以降の文がすべて実行される
Not found
```

図 6.29 switch 文の動作を見るプログラム

さて，この結果からもわかるとおり，与えられた式の値がある case に一致すると，switch 文はそれ以降の case に相当する文をすべて実行してしまいます．

しかし普通は a が入力されたら A の動作，b が入力されたら B の動作というように，それぞれ別々の動作を行う機能のほうがむしろ要求されるでしょう．そのためには，図 6.30 のプログラムに示すように，switch 文の中で break 文を利用します．

```
#include <stdio.h>
main()
{
    switch (getche()) {
     case 'a':
        puts("¥nApple");
        break;
     case 'b':
        puts("¥nBanana");
        break;
     default:
        puts("¥nNot found");
        break;
    }
}
```

実行結果 1

```
a       ……………… a を入力した
Apple   ……………… Appleだけ表示された
```

実行結果 2

```
b       ……………… b を入力した
Banana  ……………… Banana だけ表示された
```

図 6.30 break 文を用いた一般的な switch 文の使用例

break 文でループから抜け出せることについては説明しましたが，このように switch 文から抜け出すにも break 文が利用できるわけです．

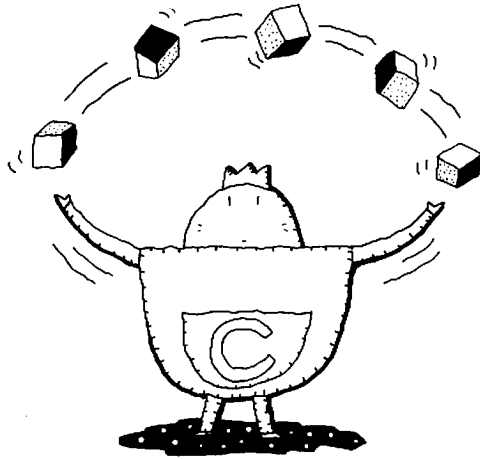
ところで，なぜ switch 文がこんなヤヤコしい構造になっているかといいますと，それは図 6.31 のプログラムのような使い方，つまり複数のラベルが同じひとつの飛び先を指定できるように考えたからということです．

```
#include <stdio.h>
main()
{
    switch (getche()) {
     case 'A':
     case 'a':
        puts("Apple¥n");
        break;
     case 'B':
     case 'b':
        puts("Banana¥n");
        break;
     default:
        puts("Not found¥n");
        break;
    }
}
```

図6.31 大文字も小文字も同じ飛び先に飛ぶ

ともあれ，switch文にはbreak文，これはもうワンセットで覚えてしまったほうがよいかもしれませんね．

7章

数値データの扱い方

ここまでの章では，数なんて計算できてあたりまえ，Cったって足し算や引き算するのにBASICと変わるわけないよーん，という安直な姿勢で説明を進めてきました．まあ確かにプログラミング言語が違っても，基本的な数の扱いにそれほど差が出るはずもありません．1+1が2になるのは宇宙の真理です．

しかし，すでに見てきたとおり，ちょっとプログラムの奥の細道をのぞいてみると，CとBASICはアレコレと違う振舞いをします．数値データの扱いにしてもしかり．単なる数値といって油断はできません．

というわけで，この章では，C言語における数値データの扱いについて，しっかりと基礎を固めておきましょう．

7 1 数値のいろいろな書き表し方

プログラミングの場では，10 進数以外の数値表現が使えると便利な場合があります．とくにグラフィックパターンを表示したり，ハードウェアを直接操作するような場合には，16 進数は欠かすことができません．

ここでは，それらのいろいろな数値表現と，その画面への出力方法について説明します．

4 種類の数値表現

MSX-C に用意されている数値の表現方法を順にあげてみると，まずごく普通の 10 進数，そして 16 進数と 8 進数があります．また，文字定数も一種の数値と考えてよいでしょう．結局のところ MSX-C では，表 7.1 に示した 4 種類の数値表現が利用できることになります．

数値表現	例	表記の方法	データの型
10進数	－1536	ごく普通の数の表記	int／unsigned
16進数	0xfa00	先頭に0xを付ける	unsigned
8進数	0175000	先頭に0を付ける	unsigned
文字定数	'A'	文字をシングルクォートで囲む	char

表 7.1 C の数値表現

● 10 進数

10 進数とは，いまさら言うまでもありませんが，普通の 10 進数字（0～9）を並べたものです．ただし C では，数字の先頭に 0（ゼロ）を付けると，後で述べる 8 進数とみなされてしまいますから注意してください．

なお，10 進数には int 型と unsigned 型の 2 種類の解釈のしかたがありますが，このことについては次の 7.2 節で説明します．

● 16進数

ハードウェアを直接操作したり，データのビットパターンを見るようなテクニカルな用途に使う数値表現といえば，この16進数で決まりです．

Cで16進数を表すには，0x(ゼロエックス)に続けて，16進数字(0〜9，a〜f)を並べます．このとき16進数字のa〜fは大文字のA〜Fでもかまいませんが，0xのxはかならず小文字で書かなくてはなりません．

● 8進数

いま，テクニカルな用途に使う数値表現といえば16進数だ！　と言い切ってしまいましたが，実はそう言えるようになったのは最近のことでして，一昔前の(C言語が誕生した)時代には，16進数よりむしろ8進数のほうがよく利用されていました．その名残からか，Cでは今でも8進数が使われることが少なくありません．

8進数を表すには，「0(ゼロ)」に続けて8進数字(0〜7)を並べます．これは一見すると10進数と間違えやすいので注意してください．

●文字定数

文字定数はシングルクォート記号で1個の文字を囲んだものです．文字定数が数値の表現だといわれると，すこし奇妙な感じですが，Cでは文字定数はその文字コードに等しい数値として扱ってかまいません．

これらの各形式で書き表した数値は，10進数とまったく同様にプログラムの中で利用することができます．図7.1に，16進数／8進数／文字定数を使ったサンプルプログラムを示します．

なお，文字定数はchar型のデータですから，printf()に直接，

```
printf("%d", 'A') ;              ……  誤り
```

という形で表示させることはできません．一般のCではこれでも問題ないのですが，関数のパラメータの型の区別に厳しいMSX-Cでは，図7.1の7行目のようにint型にキャスト(後述)する必要があります．

```
Ø: #include <stdio.h>
1: main()
2: {
3:     int  i;
4:
5:      printf("%d", Øx123);      putchar('¥n'); ……16進数を10進数表示
6:      printf("%d", Ø123);       putchar('¥n'); …… 8 進数を10進数表示
7:      printf("%d", (int)'A');   putchar('¥n'); ……文字定数を10進数表示
8:
9:      i = Øx123 + Ø123 + 'A';
1Ø:     printf("%d", i);
11: }

実行結果
291  ……………16進数 0x123は10進数に直すと291
83   ……………8 進数0123は10進数に直すと83
65   ……………文字定数'A'は A の文字コードに相当する65
439  ……………0x123+0123+'A'は10進数に直すと439
```

図 7.1　16 進数，8 進数，文字定数の使用例

16 進数や 8 進数の表示

さきほどの図 7.1 に示したプログラムでは，10 進数以外の形式で表した数値でも，printf() で画面に出力するときには 10 進数として表示されました．これは別におかしな現象ではありません．なぜなら 16 進数や 8 進数という特別な「数」が存在しているわけではないからです．

16 進数や 8 進数などは、あくまでもプログラムリスト上での数値の書き方にすぎません．そして数値がどのように書かれていても，コンパイルしてしまえば結果は一緒，コンピュータの内部では，10 進数も 16 進数も 8 進数も，みんな 2 進データに変換されています．

そのため printf() を使うときは，数値データをどの形式で画面に表示したいのか，こちらから指定してやらなくてはなりません．逆にいえば，指定の方法さえ変えれば，ある数値をどの形式でも自由に表示できるのです．

たとえば，ここまでは printf() にはかならず "%d" の指定記号を使ってきましたが，これは数値を 10 進数の形式で表示せよという意味でした．printf

()では，この10進数に加えて，16進数，8進数，文字，符号なし10進数と，合わせて5種類の数値形式が指定できます．

表7.2にそれぞれの指定記号をまとめます．

記号	意味
%d	10進形式
%x	16進形式
%o	8進形式
%c	文字形式
%u	符号なし10進形式

表7.2　printf()の表示指定記号

以下，各記号の使い方を簡単に説明しましょう．

●数値を10進数として表示——%d記号

"%d"は，すでに紹介してきたとおり，数値データを−32768～32767の範囲の10進数の形式で表示します．後で述べる"%u"との違いに注意してください．

●数値を16進数として表示——%x記号

"%x"は，数値データを16進数として表示します．ただし，このとき16進数の先頭に0xは表示されません．またMSX-Cでは，16進表示には大文字のA～Fを使いますが，コンパイラによっては小文字のa～fを表示するものもあります．

●数値を8進数として表示——%o記号

"%o"（oは小文字のオー）記号を使用すると，数値データは8進数の形式で表示されます．ただし8進数の先頭に0は付きません．

●数値を1個の文字として表示——%c記号

"%c"はちょっと特殊な形式で，数値データを文字コードと考え，それに

相当する文字1個を表示します.

これは putchar()の動作によく似ていますが, putchar()が char 型のデータを表示するのに対し, この printf()の%c 指定は, int 型(または unsigned 型)のデータを文字として表示するものです.

●数値を符号なし10進数として表示——%u 記号

"%u"は, 数値データを0～65535の範囲の正の10進数として表示します. たとえば0xffff という16進数は, %d 指定ならば－1と表示されますが, %u 指定を使うと65535になります. これは後で述べる unsigned 型のデータを表示するために用意された表示形式です.

図7.2に5種類の形式で64から80までの数値を表示するプログラムを示します.

```
#include <stdio.h>
main()
{
    int  i;

    for (i = 64; i <= 80; ++i) {
        printf("%d", i);    putchar(' ');
        printf("%x", i);    putchar(' ');
        printf("%o", i);    putchar(' ');
        printf("%c", i);    putchar(' ');
        printf("%u", i);    putchar('¥n');
    }
}


64 40 100 @ 64
65 41 101 A 65
66 42 102 B 66  ←—— 実行結果
:  :  :   : :
79 4F 117 O 79
80 50 120 P 80
```

図7.2 printf()による5種類の形式の数値表示

7-2 数値データの型と，その利用法

Cで扱う数値データは，ある決まった「**データの型**」を持っています．これは基本的には扱う数値範囲の違いを区別するのですが，数値の代入や比較演算を行う際にも，データ型の違いはプログラムにさまざまな影響を与えます．

この節では，そのデータ型とはどういうものか，さらに個々のデータ型の特徴について見ていきましょう．

データの型とは何か

最初に簡単な実験を1つ．まず，16進数の0x9000と0x3000という2つの数を考えてください．これらを10進数形式で表示してみると，図7.3のように，0x9000は−28672というマイナスの数値，0x3000は12288というプラスの数値になっています．

```
#include <stdio.h>
main()
{
    puts("¥nØx9ØØØ == ");
    printf("%d", Øx9ØØØ);
    puts("¥nØx3ØØØ == ");
    printf("%d", Øx3ØØØ);
}
```

```
Øx9ØØØ == -28672
Øx3ØØØ == 12288
```
← 実行結果

図7.3　0x9000と0x3000の値を確かめる

さてそこで，この2つの16進数の大小を比較し，どちらが大きいかを表示させてみます．図7.4がその実行結果です．

```
#include <stdio.h>
main()
{
    if (Øx9ØØØ > Øx3ØØØ)
        puts("Øx9ØØØ > Øx3ØØØ");
    else
        puts("Øx9ØØØ < Øx3ØØØ");
}
        ┌──────実行結果
        ↓
Øx9ØØØ > Øx3ØØØ  ……………0x9000は0x3000より大きい？
```

図7.4　この結果は変ではないか？

すると結果は上のように，「0x9000 > 0x3000」と表示されたと思います．マイナスの数であるはずの0x9000のほうが，プラスの数であるはずの0x3000よりも大なりとは，これいかに．

その理由は，Cでは16進数をunsigned型のデータとして扱う約束になっているからです．unsigned型とは，別名「**符号なし整数**」とも呼ばれるもので，int型のデータとは基本的に種類の異なるデータ型とみなされます．

●データの型とはデータを扱う考え方である

MSX-Cでは整数を16ビットの2進データで表現しますが，この2進データは符号付きの整数であると考えることも，符号なしの整数と考えることもできます．

1001000000000000	符号付き整数と考えれば−28672
(16進では0x9000)	符号なし整数と考えれば36864

そしてCでは，データを符号付きの整数として扱う場合，それをint型のデータと呼び，符号なしの整数として扱う場合，それをunsigned型のデータと呼ぶのです．

unsigned 型と int 型のデータの違いの例を 1 つ図 7.5 に示します．ここでは，さきほどの 0x9000 と 0x3000 を，それぞれのデータ型の変数に代入した上で比較しています．

```
#include <stdio.h>
main()
{
    int      i1, i2; ··············int型の変数i1とi2を用意
    unsigned u1, u2; ··············unsigned型の変数u1とu2を用意

    i1 = 0x9000;
    i2 = 0x3000;
    if (i1 > i2) ··············unsigned型で比較する
        puts("int ------- 0x9000 > 0x3000¥n");
    else
        puts("int ------- 0x9000 < 0x3000¥n");

    u1 = 0x9000;
    u2 = 0x3000;
    if (u1 > u2)
        puts("unsigned -- 0x9000 > 0x3000¥n");
    else
        puts("unsigned -- 0x9000 > 0x3000¥n");
}


int ------- 0x9000 < 0x3000
unsigned -- 0x9000 > 0x3000  ←──── 実行結果
```

図 7.5　int 型の比較と unsigned 型の比較の違い

結果はごらんのとおり，同じ 2 つの数値でも，int 型では「0x9000　<　0x3000」，unsigned 型では「0x9000 > 0x3000」とみごとに違う回答が得られました．このように，コンピュータが数値をどのようなものと考え，どのように扱うかを区別するのが「データの型」なのです．

3 種類のデータの型

ここで，MSX-C に用意されている，基本的な数値データの型を紹介しておきましょう．表 7.3 にその一覧を示します．

データ型	扱う数値の範囲	サイズ
int型	－32768～32767	16ビット(２バイト)
unsigned型	0～65535	16ビット(２バイト)
char型	0～255	8ビット(１バイト)

表 7.3　MSX-C の数値データの型

他の C コンパイラでは，これら以外にも小数点以下まで表現できる実数型(float 型，double 型)や，約 20 億までの大きな整数が表現できる倍精度整数型(long 型)が利用できる場合もありますが，MSX-C には上の 3 種類のデータ型しか用意されていません(ただし，別売の「MSX-C ライブラリ」を併用すると，MSX-C でも実数や倍精度整数が使えるようになります)．

● int 型のデータ

int 型のデータは，MSX-C で使われる最も基本的な数値データです．これはビット数でいうと 16 ビットのサイズを持ち，そのため int 型のデータを変数に記憶させる場合には 2 バイトのメモリを必要とします．

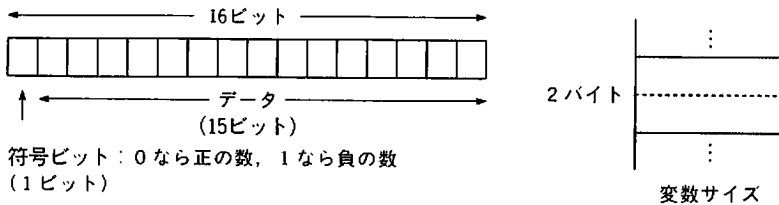



図 7.6　int 型のデータ構造

int 型のデータには以下のものがあります．

- －32767～32768 の範囲の 10 進定数
- int 型の変数
- (int)演算子でキャストした任意のデータ

3 番目にあげた「(int)」は，キャスト演算子と呼ばれるものの 1 つで，型

の名前をカッコで囲むことにより表されます．キャスト演算子を頭に付けると，本来はほかの型であるデータを，指定した型のデータとして強制的に扱うことができます．

たとえば，16 進数は本来は unsigned 型のデータなのですが，図 7.7 のようにキャスト演算子(int)を使用すると，コンパイラはこれを int 型のデータとして扱ってくれます．なお，このようにキャスト演算子を使ってデータの型を変換することを，そのデータ型に「**キャストする**」といいます．

```
#include <stdio.h>
main()
{
    if ((int)0x9000 > (int)0x3000)
        puts("0x9000 > 0x3000");
    else
        puts("0x9000 < 0x3000");
}

          ┌────実行結果
          ↓
0x9000 < 0x3000  ……………0x9000と0x3000がint型として比較された
```

図 7.7　キャスト演算子(int)の効果

● unsigned 型のデータ

unsigned 型は，すでに述べたように，int 型と同じ 16 ビット(2 バイト)の数値データを符号なしの整数として扱うためのデータ型です．

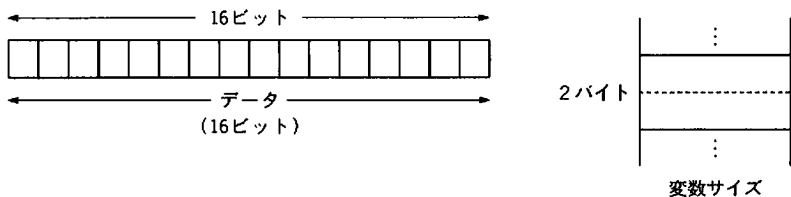



図 7.8　unsigned 型のデータ構造

unsigned 型のデータには以下のものがあります．

- 32768 以上の 10 進定数(32768〜65535)
- 16 進定数
- 8 進定数
- unsigned 型の変数
- (unsigned)演算子でキャストした任意のデータ

unsigned 型のデータを利用したい場合は，かならず変数も unsigned 型として宣言してください．int 型と unsigned 型が混乱すると，わけのわからない結果が出てきて何時間も頭をひねることがあります．試しに図 7.9 のプログラムを実行してみてください．

```
#include <stdio.h>
main()
{
    int  i;

    for (i = 0; i <= 60000; ++i) {
        printf("%d", i);
        putchar(' ');
    }
}

実行結果
何もおきない
```

図 7.9 int 型の変数で unsigned 型の 60000 はカウントできない

このプログラムは，0 から 60000 までの数値を表示させるつもりだったのですが，いざ実行してみると，なにもしないで終了してしまいました．その原因は，ループ変数 i が int 型として宣言されていたからです．

MSX-C が異なる型のデータを比較する場合，変数と定数ではつねに変数のデータ型のほうが優先されます．ですから，この場合「i <= 60000」という条件判断は int 型データの比較とみなされ，上の for 文はコンパイラに次のように解釈されてしまいます．

```
for (i = 0; i <= -5536; ++i) {
        ……        (60000 を int 型に直すと -5536)
```

これは，変数 i の値を 0 から増加させつつ-5536 まで繰り返すループ，いわゆる「おまえはすでに終わっている」ループですから，当然何も表示されないわけです．

数値をこのように unsigned 型のデータとして扱いたい場合には，変数も次のように unsigned 型として宣言しなければなりません．

```
unsigned  i;        ← unsigned 型として宣言する
```

● char 型のデータ

char 型は，数値を 8 ビット(1 バイト)サイズの符号なし整数として扱うためのデータ型です．

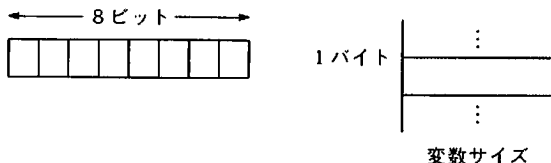



図 7.10　char 型のデータ構造

char 型のデータには以下のものがあります．

- 文字定数
- char 型の変数
- (char)演算子でキャストした任意のデータ

これは MSX-C に用意されたデータ型の中で，ただ 1 つデータの内部表現のサイズが異なるため(ほかのデータ型はすべて 2 バイトです)，利用にあたっては注意が必要です．

たとえば putchar()は char 型データ専用の表示関数ですから，これで int

型のデータを表示しようとしてもうまくいきません．

```
putchar(65) ;                 …… 誤り(65 は int 型)
putchar('A'+1) ;              …… 誤り('A'+1 は int 型)
```

その逆に printf()は int 型のデータしか扱えませんから，次のように printf()に char 型の文字定数を与えても，望んだ結果は得られません．

```
printf("%d", 'A') ;           …… 誤り('A'は char 型)
```

これらの例のように，要求されているものと異なる型のデータを使う場合には，キャスト演算子を使ってデータの型を目的の型に強制的に合わせる必要があります．上の 3 つの誤った関数呼び出しを正しいデータ型にキャストすると，次のようになります．

```
putchar((char)65) ;           …… 正しい(文字 A が表示される)
putchar((char)('A'+1)) ;      …… 正しい(文字 B が表示される)
printf("%d", (int)'A') ;      …… 正しい(数値 65 が表示される)
```

MSX-C ではデータの型の間に表 7.4 に示す優先順位があり，数式のデータ型は，その中に含まれる一番優先順位の高いデータに合わされます．

強い	unsigned型の変数
↑	int型の変数
	char型の変数
	unsigned型の定数(16進数，8進数，32768以上の10進数)
↓	int型の定数(−32767～32768の範囲の10進数)
弱い	char型の定数(文字定数)

表 7.4 データの型の優先順位

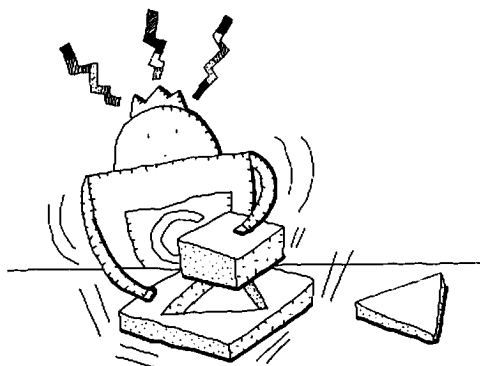
簡単にいうと，定数よりも変数のほうが強く，char 型よりも int 型，int 型よりも unsigned 型のほうが強という法則があるわけですが，いちいち考えるのが面倒ならば，データの型が混合していて結果がアヤシくなりそうな場合には，かならずキャスト演算子を使ってしまうというのも手かもしれません．

データ型の整合性チェック

ここでは，前項の説明を受けて，データの型の誤りから起こるバグを未然に防ぐ方法について述べます．これはデータの型を正しく扱っている限り無用の知識ですから，もし読むのが面倒ならば読みとばしていただいてもかまわないでしょう．

MSX-C では関数に引き渡すパラメータのデータ型の区別に厳しく，これを間違えると正常な動作は望めません．しかしそれにも関わらず，MSX-C コンパイラは関数パラメータのデータ型の誤りに対してはエラーチェックを行ってくれません．

そのため，コンパイルは正常に終了したのに，いざ実行してみると思いどおりの結果が得られないという事態も起こりえます．とくに，他の C で動いているプログラムを MSX-C に移植する場合などには，しばしばこの問題が影響してきます．一般の C では int 型と char 型はほとんど違いを意識せずに使えるため，たいていの場合両者は混ぜこぜに使われているからです．

そこで MSX-C には，FPC(ファンクション・パラメータ・チェッカ)というサポートツールが用意されました．FPC コマンドは，いま作成しているプログラム中の関数呼び出しと，別ファイルに登録してある関数の正しい使用法を照らし合わせ，誤った関数呼び出しの行われている場所をすべて示してくれます．

FPC コマンドによる関数呼び出しのエラーチェック方法を，実際にバグを含んだプログラムを例にして説明しましょう．

まず，次のプログラムを"AIUEO.C"というファイル名で作成してください．これは「ア」から「ン」までの 45 文字を表示するプログラムなのですが，コンパイルは正常に行われるのに，実行してみるとうまく動いてくれません（みなさんも試してみてください）．

```
#include <stdio.h>
main()
{
    int   i;

    for (i = Ø; i < 45; i++)
        putchar('ｱ'+i);
}
```

図 7.11　関数へのパラメータの型を間違えた例

こんなときは FPC の出番です．FPC でプログラムをチェックするには，いったんソースプログラムを T コードファイルに変換する必要があります．つまり，次のように CF コマンドを実行して"AIUEO.TCO"を作るわけです．

```
A>cf aiueo ⏎ ……………"AIUEO.C"をCFにかける
MSX-C ver 1.1ØP (parser)
Copyright (C) 1987 by ASCII Corporation
complete
A>dir aiueo.tco ⏎ ……………"AIUEO.TCO"の存在を確認する
AIUEO    TCO    256 89-Ø1-Ø8 Ø6:3Øa
       1 file  366592 byte free
A>
```

図 7.12　T コードファイルを生成する

次に，この T コードファイルを，あらかじめ用意してある"LIB.TCO"と一緒に FPC コマンドにかけます．"LIB.TCO"は，MSX-C の全ライブラリ関数

の正しい使用法が記録されたTコード形式のファイルです．

ここで，もし何も問題がなければ，FPCは単に「complete」とだけ表示するでしょう．しかし上のプログラムの場合は，次のようなエラーメッセージが出てきます．

```
A>fpc aiueo lib ⏎ ……………"AIUEO.TCO"と"LIB.TCO"を照会(拡張子は省略)
MSX C function parameter checker ver 1.10s
in <aiueo.TCO> "main" calls "putchar" : 1th argument conflict
comlete
```

図7.13　FPCのエラーメッセージ

このメッセージを日本語に意訳すると「<aiueo.TCO>というファイルの中で，main()がputchar()を呼び出しているが，その1番目の引数の型が誤っている」ということになります．そこで元のソースプログラム"AIUEO.C"を見てみると，なるほどっ，main()の中でputchar()に与えているパラメータ「'ア'+i」は，char型ではなくint型ではありませんか(前項で述べたように，char型定数+int型変数はint型になります)．

というわけで，キャスト演算子を使って，プログラムを次のように訂正します．これをコンパイルして実行すれば，最初の意図どおりにアからンまでの45文字が表示されるはずです．メデタシめでたし．

```
#include <stdio.h>
main()
{
    int   i;

    for (i = 0; i < 45; i++)
        putchar((char)('ｱ'+i));
}
                'ア'+iをchar型キャストする
```

図7.14　FPCのメッセージに従って修正したプログラム

FPCの話が出たついでに，このコマンドによる関数呼び出しの正誤チェックについて，もう少し述べておきます．

一般にFPCのエラーメッセージは次のような形式で出力されます．わざわざファイル名(ここでは<aiueo.TCO>)まで表示するのは，後で説明する「プログラムの分割コンパイル」を行う際に，どのファイルの中でエラーが発生したかを見分けるためです．

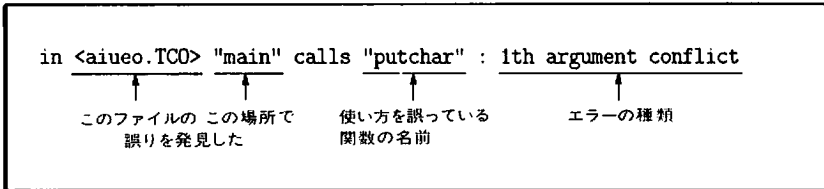



図7.15 FPCのエラーメッセージの読み方

通常のライブラリ関数を使ったプログラムでは，FPCコマンドで発見できるエラーには全部で3種類のものがあり，それぞれ以下のようなメッセージが表示されます．

①「…… : conflicting number of arguments」

パラメータの個数が誤っていることを示します．ただしprintf()などの可変パラメータ関数(パラメータの個数がいくつでもよい関数)については，当然ながらこのエラーチェックは行われません．

②「…… : ?th argument conflict」(?の部分には数字がはいる)

すでに述べたとおり，このエラーメッセージはパラメータのデータの型が誤っていることを示します．

ただしprintf()などの可変パラメータ関数に対しては，FPCはデータ型のチェックを行いません．つまり，誤ってprintf()にchar型のパラメータを与えても，FPCはそれをエラーと判断してくれないのです．

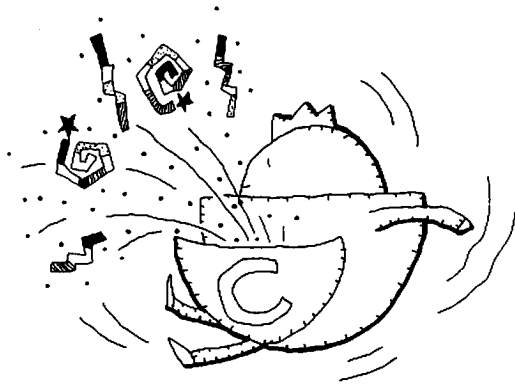
ですからprintf()には，プログラマの責任において正しいint型のデータを与えてやる必要があります．まったくもって言語同断な話ですが，FPCコマンドの仕様がそうなっている以上しょうがありません．

③「……　：　undefined」

これは未定義の関数が使われていることを示します．いまの段階でこのエラーが発生するということは，MSX-Cに用意されていないライブラリ関数を使用したか，あるいは関数名のスペリングを誤ったかのどちらかでしょう．もう少し先に進むと，プログラムの分割コンパイルを行うときに，このエラーメッセージが意味を持つことになります．

以上，FPCコマンドでチェックできるエラーの中で，ライブラリ関数の使用法に関連するものを紹介しました．これ以外のFPCの機能は，それぞれ必要となった時点で改めて説明することにします．

8章

画面表示を工夫する

Cのプログラミングに関する知識もだいぶ蓄積が増えました．内容が充実してきたところで，そろそろ結果の見せ方にも気をつかってみましょう．ここまでは，文字の表示にしても数値の表示にしても，単に画面に出てくればいいやぐらいの考えでプログラムを作ってきましたが，やはりカッコよい表示を行うプログラムは，格も違って見えます．

BASICには，CLS命令やLOCATE命令など，画面のコントロールを行う命令がいくつか用意されていましたが，MSX-Cには，それに相当する機能のライブラリ関数はありません．しかし，コントロール文字やエスケープシーケンスを画面に出力することによって，これらの機能は簡単に実現できるのです．

8 1 フォーマット指定付きの数値表示

printf()を使うと，16進数や8進数などの形式で数値データを表示できることは前章で述べましたが，printf()の実力はそれだけではありません．数値の桁ぞろえや，数値以外の文字列を添える方法など，数値をさらに見やすい形式で表示する機能も用意されています．

ここでは，そのようなprintf()のちょっと高度な使い方を紹介することにしましょう．

数値の桁ぞろえ表示

数値を桁ぞろえして表示するprintf()の機能には，3つの種類があります．それを表8.1にまとめてみました．

なお，この表には10進形式の数値表示(%d指定)に対する桁ぞろえの方法だけを示しましたが，これ以外の表示形式(%x，%o，%c，%u)に対しても，同様な桁ぞろえ指定が可能です．

機　能	指定記号	例	表示結果
数値の右ぞろえ	%桁数d	printf("%6d", 123);	···123
数値の左ぞろえ	%−桁数d	printf("%−6d", 123);	123···
左端を0で埋める	%0桁数d	printf("%06d", 123);	000123

表8.1　printf()の桁ぞろえ表示機能

以下，それぞれの機能について簡単に説明します．

●数値の右ぞろえ表示

printf()で10進数を表示するとき，%とdの間に桁数を入れて，

%桁数d　　(桁数は任意の10進数)

という指定を行うと，その桁数の中に数値を右ぞろえに表示できます．このとき，左側の余った桁は空白で埋められます．また，この右ぞろえ表示は，%x，%o，%c，%uの各表示形式に対しても同様に指定することができます．

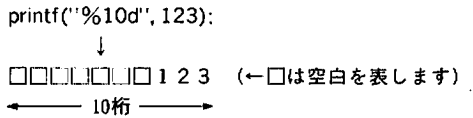



図8.1　右ぞろえの表示形式

●数値の左ぞろえ表示

桁数の前にさらにマイナス記号を付け，

%－桁数d　　(桁数は任意の10進数)

という指定を行うと，数値をその桁数の中に左ぞろえで表示できます．これもやはり%d，%x，%o，%c，%uの各形式に対して指定可能です．

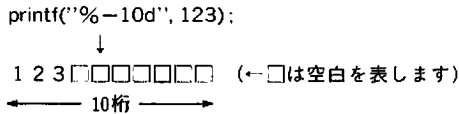



図8.2　左ぞろえの表示形式

●右ぞろえして左端を0で埋める

桁数の前に数字の0を付け，

%0桁数d　　(桁数は任意の10進数)

という指定を行った場合は，数値を右ぞろえにして，さらに余った左側の桁

を数字の 0 で埋めることができます．やはり %d，%x，%o，%c，%u のどの形式に対しても指定可能です．

```
printf("%010d", 123);          (← 010 と書いても 8 進数
        ↓                       ではないことに注意)
0000000123
←―― 10桁 ――→
```

図 8.3　右ぞろえして左端を 0 で埋める表示形式

図 8.4 に，アルファベットの文字コードを 10 進，16 進，8 進，の各形式で表示するプログラムを示します．

```
#include <stdio.h>
main()
{
    int  i;

    puts(" ﾓｼﾞ   10ｼﾝ 16ｼﾝ  8ｼﾝ¥n");
    puts(" ---- ---- ---- ----¥n");

    for (i = 'A'; i <= 'Z'; ++i) {
        printf("%5c", i);
        printf("%5d", i);
        printf("%5x", i);
        printf("%5o", i);
        putchar('¥n');
    }
}
```

```
ﾓｼﾞ  10ｼﾝ 16ｼﾝ  8ｼﾝ
---- ---- ---- ----
   A   65   41  101
   B   66   42  102      ←――実行結果
   :    :    :    :
   X   88   58  130
   Y   89   59  131
   Z   90   5A  132
```

図 8.4　%d 以外の形式に対しても右ぞろえの指定ができる

フォーマット指定付きの数値表示

ここまで，printf()を単独の数値を表示するために用いてきましたけれども，実際にはprintf()にはもっと便利な使い方があります．まず，数値以外の文字を一緒に表示することができます．さらに複数の数値を1つのprintf()で表示することもできます．ここではその2つの方法について説明しましょう．

●表示フォーマットの指定

printf()の1番目のパラメータである文字列は，いままで見てきたように，数値表示のいろいろな形式(フォーマット)を指定するために使われます．そのため，この文字列は一般にフォーマット文字列と呼ばれます．

フォーマット文字列の中には，図8.5のプログラムのように，表示形式の指定(ここでは%d)と通常の文字を混在させることができます．

```
#include <stdio.h>
main()
{
    printf("1 ﾈﾝ ﾊ %d ｼﾞｶﾝ ﾃﾞｽ", 24 * 365);
}


1 ﾈﾝ ﾊ 8760 ｼﾞｶﾝ ﾃﾞｽ ←――――実行結果
```

図8.5　数値と文字を1つのprintf()で一緒に表示する

この場合，printf()は指定記号%dの部分だけを表示したい数値に置き換えて，残りの文字はそのまま画面に出力します．

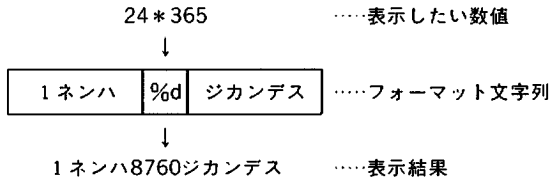



図 8.6　%d と数値の置き換えのしくみ

もちろん，%d だけではなく %x，%o，%c，%u の各記号や，それに桁数を加えた %6d や %04x などの指定記号も自由に使ってかまいません．

フォーマット文字列のそれ以外の部分には，puts() で表示する通常の文字列とまったく同じように文字を並べることができますが，ただパーセント記号 % だけは特別です．パーセント記号は，printf() にとって，数値形式の指定の始まりという特殊な意味を持つため，実際に「%」という文字を printf() で表示したい場合には，図 8.7 のように記号を 2 つ重ねて %% と書いてください．

```
#include <stdio.h>
main()
{
    printf("ｼｮｳﾋ ｾﾞｲ ﾊ %d %% ﾀﾞﾖ", 3);
}


ｼｮｳﾋ ｾﾞｲ ﾊ 3 % ﾀﾞﾖ ←———— 実行結果
```

図 8.7　printf() で % という文字を表示する方法

● 複数個のデータを 1 つの printf() で表示する

また printf() は一度に複数の値を表示することも可能です．その場合は，フォーマット文字列の中に %d を必要なだけ並べておきます．たとえば図 8.8 のプログラムでは，39，402，39×402 の 3 つの数を printf() で表示しています．

```
#include <stdio.h>
main()
{
    printf("%d * %d == %d", 39, 402, 39*402);
}



39 * 402 == 15678 ←――― 実行結果
```

図 8.8　複数個のデータを 1 つの printf() で表示する

このとき，フォーマット文字列中の指定記号は，左から順に表示したいパラメータで置き換えられます．

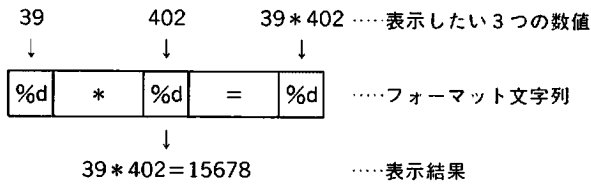
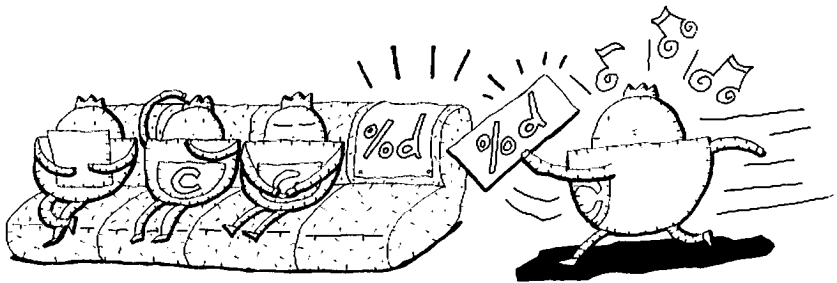



図 8.9　複数個のデータと%d の置き換えのしくみ

8 2 コントロール文字による画面制御

MSXには，コントロール文字と呼ばれる特殊な文字がいくつか用意されています．これらは画面に出力しても目に見える文字としては表示されず，その代わりに画面クリアやカーソル移動などの画面制御を行います．

特別な記号を持つコントロール文字

Cでは，このコントロール文字も，1つの文字として表すことができます．ただし，なにしろ目に見えない文字ですから，これをプログラムとして打ち込むときには，ちょっとした表記の工夫が必要です．

たとえば，改行させるためのニューライン文字¥nはすでに紹介しましたが，この¥nの正体が，実は文字コード10に相当するコントロール文字なのです．つまり，「改行」という目に見えない存在を，キーボードから打ち込める現実の文字として表す工夫が，¥nという記号だったわけです．

MSX-Cでは，この¥nを含めて，表8.2に示す7種類の記号がコントロール文字を表すために利用できます．

記号	10進文字コード(16進/8進)	機能
¥a	7(0x07/007)	ビープ音を鳴らす
¥b	8(0x08/010)	カーソルを1文字左に移動させる
¥t	9(0x09/011)	カーソルを8の倍数の位置に進める
¥n	10(0x0A/012)	カーソルを次の行の先頭に進める
¥v	11(0x0B/013)	カーソルを画面左上に移動させる
¥f	12(0x0C/014)	画面をクリアする
¥r	13(0x0D/015)	カーソルを現在行の先頭に戻す

表8.2　特別な記号で表されるコントロール文字

これらのコントロール文字を画面に出力すると，上の表に示した動作が実行されます．文字として出力されるならば，どんな方法でもかまいません．たとえば画面をクリアする¥f(文字コード12)には，次の3つの出力方法が考えられます．

```
putchar('¥f') ;
puts("¥f") ;
printf("%c", 12) ;
```

どれも画面クリアを行う

文字コードで表すコントロール文字

MSXで利用できるコントロール文字は，上で述べた7種類だけではなく，このほかに表8.3にあげたものが用意されています．ただしCでは，これらについては¥nや¥tのような記号を定めていません．

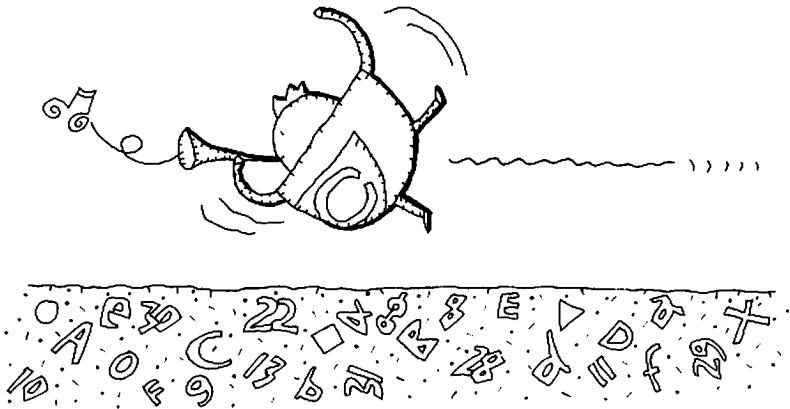
10進文字コード(16進/8進)	機　能
1(0x1/01)	グラフィック文字の始まりを示す
27(0x1b/033)	エスケープシーケンスの始まりを示す
28(0x1c/034)	カーソルを1文字右に移動させる
29(0x1d/035)	カーソルを1文字左に移動させる
30(0x1e/036)	カーソルを1文字上に移動させる
31(0x1f/037)	カーソルを1文字下に移動させる
127(0x7f/0177)	カーソルを1文字左に戻し，文字を消去する

表8.3　特別な記号が付けられていないコントロール文字

では，これらのコントロール文字はプログラム中で利用できないのかというと，そんなことはありません．実はCでは¥xの後に16進文字コードを続けることによって，どんな文字でも表すことができるからです．たとえばカーソルを1文字下へ移動させるコントロール文字(文字コード0x1f)は，¥x1fと表現できます．

puts("ナ¥x1f ナ¥x1f メ");　　　……　「ナナメ」と斜めに表示する

文字コードは 16 進数以外に 8 進数を使うことも可能です．その場合は単独の¥の後に 8 進数で表した文字コードを続けます．たとえば¥x1f は次のように 8 進コードで¥37 とも表せるのです．

puts("ナ¥37 ナ¥37 メ");　　　……　上と同じ結果になる

8.3 エスケープシーケンス

前節で述べたコントロール文字の中に，¥33(¥x1b)というものがありました．これは少々変わった機能を持っていまして，単独で出力しても画面には何もおきません．しかし，この文字に続けて，ある特定の文字の並びを出力すると，いろいろな効果が得られます．

エスケープシーケンスを使ってみる

たとえば，¥33, y, 4, の 3 文字をこの順に出力すると，カーソルがアンダーライン「_」の形に変わってしまいます．

```
#include <stdio.h>
main()
{
    puts("¥33y4");
}
```

実行結果
カーソルの形が「 _ 」になる

図 8.10 エスケープシーケンスの使用例

このように¥33 の文字で始まる文字の並びを，一般にエスケープシーケンスと呼びます．MSX には，以下の表に示す 16 種類の画面制御用エスケープシーケンスが用意されています．

これらのエスケープシーケンスは，実際に使ってみないと動作を理解しにくいかもしれません．以下に続く項で，いくつか具体的な使い方を紹介することにしましょう．

エスケープシーケンス	機　能
¥33J	カーソルの位置から画面の最後までをクリア
¥33K	カーソルの位置から行の最後までをクリア
¥33L	カーソルの位置に1行差し込み
¥33M	カーソルの位置にある1行を削除する
¥33Y??	カーソルを??で指定した位置に移動
¥33x4	カーソルの形を「■」に変える
¥33x5	1文字ごとのカーソル表示を止める
¥33y4	カーソルの形を「_」に変える
¥33y5	1文字ごとにカーソルを表示させる
¥33A	カーソル上移動←¥36(¥x1E)と同じ
¥33B	カーソル下移動←¥37(¥x1F)と同じ
¥33C	カーソル右移動←¥34(¥x1C)と同じ
¥33D	カーソル左移動←¥35(¥x1D)と同じ
¥33E	画面クリア　←¥f と同じ
¥33H	カーソルを画面左上に移動←¥v と同じ
¥33j	画面クリア　←¥f と同じ

表 8.4　MSX の画面制御エスケープシーケンス

カーソル位置の指定

エスケープシーケンス「¥33Y??」を使うと，BASIC の LOCATE 命令と同様にカーソルの位置を指定できます．ただし「?」の部分には，それぞれ y 座標と x 座標を指定する文字がひとつずつはいります．

座標を指定する文字は，その文字コードが(座標+20h)に相当するものです．たとえば座標 3 を指定するのは文字コード 23h の「#」ですし，座標 6 ならば文字コード 26h の「&」ということになります．

ですから，次のように"¥33Y #&"という文字列を出力すれば，BASIC の「LOCATE　6，3」と同じ効果が得られます(エスケープシーケンスの座標指定の順番は，y 方向が先，x 方向が後と LOCATE 命令とは逆なので注意してください)．

```
#include <stdio.h>
main()
{
    puts("¥33Y#&");
    puts("I'm here");
}
```

実行結果
座標(6,3)の位置に「I'm here」と表示

図8.11　エスケープシーケンスによるカーソル位置の指定

エスケープシーケンスは，どんな手段を使っても正しい順番で文字を送りさえすれば，うまく働いてくれます．次のようにputchar()で1文字ずつ出力してもよいのです．

```
putchar('¥33');
putchar('Y');
putchar('#');
putchar('&');
```

あるいは，printf()の%c指定を使って，次のように書くこともできます．この方法は，いちいち文字コードの表を見る必要がなくて，いちばん便利かもしれません．

```
printf("¥33Y%c%c", 0x20 + 3, 0x20 + 6);
```

行の挿入と削除

エスケープシーケンス「¥33L」は，現在カーソルが置かれている行以降を1行ずつ下へずらし，カーソルの位置に空行をあける働きがあります．このとき，もともと画面のいちばん下にあった行は消されてしまいます．

なお，このエスケープシーケンスを出力しても，カーソルは画面上の同じ位置，つまり挿入された空行の上にとどまります．

1 行挿入の図

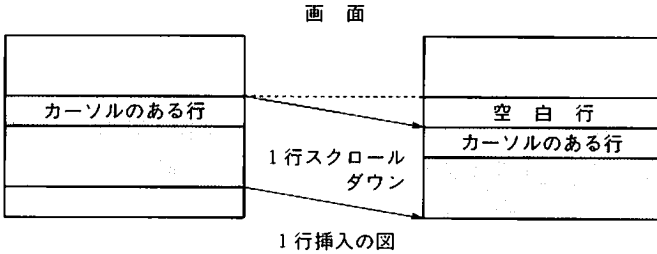



図8.12 エスケープシーケンス「¥33L」を出力したときの動作

このシーケンスは，画面全体をスクロールダウンさせたい場合にも利用できます．それには次のように，カーソルを画面の最上行に移動させ，そこで1行挿入を行えばよいのです．

```
puts("¥v¥33L¥33L¥33L");
```

「¥33L」と正反対の動作をするのが，「¥33M」のエスケープシーケンスです．こちらは現在カーソルが置かれている行を削除し，以下に書かれていた内容を全体的に1行ずつ上へ繰り上げます．その結果，画面の最下行は空白行となります．この場合も，やはりカーソルは画面上の同じ位置にとどまります．

1 行削除の図

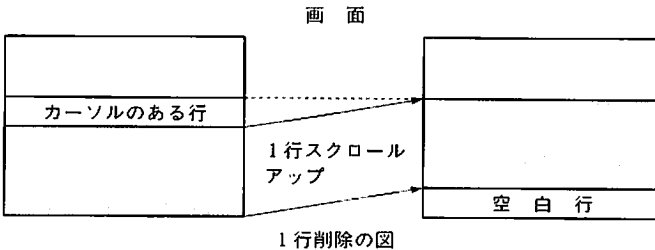



図8.13 エスケープシーケンス「¥33M」を出力したときの動作

カーソルのちらつき防止

MSX-Cのプログラムで文字を表示するとき，画面上でカーソルがチラチラ点滅するのが非常に気になることがあります．

たとえば次に示すプログラムを実行してみてください．これは矢印を画面の左端からスイーッと動かすプログラム，……のはずだったのですが，実際にはカーソルのちらつきが邪魔をして，とても見ていられません．

```
#include <stdio.h>
main()
{
    int  i;

    for (i = 1; i <= 30; ++i)
        puts(" -->¥b¥b¥b");
}
```

図8.14　カーソルがチラついて見づらい例

これは，MSX本体のカーソル表示機能が通常はONになっているため，文字をひとつ表示するごとにかならずカーソルも一緒に表示されてしまうからです．

その邪魔なカーソルを消すエスケープシーケンスが「¥33x5」です．プログラムの先頭で，

```
puts("¥33x5");
```

というシーケンスを出力しておくと，カーソル表示機能がOFFに設定され，文字がちらつくことはなくなります．

このエスケープシーケンスは，初めに1回出力しておけばプログラムが終了するまで効果が続きます．また，カーソル表示機能をOFFにしておいても，キーボード入力の際にはカーソルは自動的に表示されますから，機能のON／OFFをいちいち切り換える必要はありません．

8-4 そのほかの話題

画面表示に関して，ここまでの説明で述べきれなかった話題がいくつか残っています．最後にそれらをまとめて紹介しておきましょう．これでもう表示できない文字は1つもなくなるはずです．

グラフィック文字の表示

MSXには，グラフィック文字と呼ばれる特殊な文字が存在します．これは画面の上では1個の文字なのですが，表示するためには2バイトのデータを出力しなければなりません．

次の表にグラフィック文字を表示するためのグラフィックシーケンス一覧を示します．見ておわかりのとおり，かならず¥1のコントロール文字で始まり，さらにもう1つ文字が続く形式になっています．

これらのグラフィック文字は，見かけは1文字でも実際には2バイトのデータですから，文字定数とすることはできません．たとえば，πという文字を次のようにputchar()で表示しようとするとコンパイルエラーになります．

```
putchar('π');      ←  誤り
```

したがって，これは次のように1バイトずつに分けて出力するか，

```
putchar('¥1');
putchar('P');
```

あるいはputs()を使って次のように書く必要があります．

```
puts("¥1P");
```

シーケンス	文字(VRAMコード)	シーケンス	文字(VRAMコード)
¥1@	(00h)	¥1P	π(10h)
¥1A	月(01h)	¥1Q	┴(11h)
¥1B	火(02h)	¥1R	┬(12h)
¥1C	水(03h)	¥1S	┤(13h)
¥1D	木(04h)	¥1T	├(14h)
¥1E	金(05h)	¥1U	┼(15h)
¥1F	土(06h)	¥1V	│(16h)
¥1G	日(07h)	¥1W	─(17h)
¥1H	年(08h)	¥1X	┌(18h)
¥1I	円(09h)	¥1Y	┐(19h)
¥1J	時(0Ah)	¥1Z	└(1Ah)
¥1K	分(0Bh)	¥1[	┘(1Bh)
¥1L	秒(0Ch)	¥1¥	×(1Ch)
¥1M	百(0Dh)	¥1]	大(1Dh)
¥1N	千(0Eh)	¥1^	中(1Eh)
¥1O	万(0Fh)	¥1_	小(1Fh)

注：VRAMコードとは，VRAMに直接データを書き込んで表示する場合に用いるコードです．この表にあげられていない通常の文字については，文字コード＝VRAMコードです．

表8.5 グラフィック文字を表示するためのシーケンス

クォート記号や¥記号の表示

「¥」で始まる特殊な文字表現に次の3種類があることは，すでに見てきました．

①とくに記号の定められたコントロール文字
(¥a，¥b，¥t，¥n，¥v，¥f，¥r)
② 16進数による文字コード表現
(「¥x」に続く16進数)
③ 8進数による文字コード表現
(「¥」に続く8進数)

ここでさらに4番目として，次のような場合を付け加えましょう．

④¥の後ろの文字が上の①～③のどれにも相当しない場合，
¥？という表現は「？」の位置の1文字を表す

つまり¥の後ろが，a，b，t，n，v，f，t ではなく，x に続く 16 進数でもなく，8 進数でもない場合，それは¥の後ろに書かれた文字そのものを表すのです．たとえば「¥Z」は「Z」という文字を表すことになります．

もっとも，Z が¥Z と書けたからといって，何もうれしくありません．この表現が役にたつのは，プログラム上で特別な役割を持つ文字の意味を打ち消し，その文字自身をデータとして扱いたい場合です．

まず，puts()で表示する文字列の中にダブルクォートを含めたいときがあります．通常はダブルクォートが出てくると，そこで文字列は終わりとみなされてしまいますが，¥記号を付けると，ダブルクォート記号をデータとして文字列中に含められます．

また，シングルクォート自身を文字定数としたい場合があります．一般的な C では「'''」という書き方はエラーとなるため，やはりこの場合も¥記号を付けてシングルクォート記号自身を表す必要があります．

3 番目は，記号「¥」を文字データとしたい場合です．このときも¥¥と書くことで，最初の¥が次の¥の意味を打ち消して，¥という文字自身を表すことができます．

```
#include <stdio.h>
VOID  main()
{
    puts("ｺﾚﾊ ¥"ﾚﾝｼｭｳ¥" ﾃﾞｽ¥n");
    putchar('¥'');
    putchar('O');
    putchar('K');
    puts("ｵﾈﾀﾞﾝ ﾊ ¥¥36Ø ﾅﾘ");
}



ｺﾚﾊ "ﾚﾝｼｭｳ" ﾃﾞｽ
'OK                    ←――――実行結果
ｵﾈﾀﾞﾝ ﾊ ¥36Ø ﾅﾘ
```

図8.15　クォート記号と¥記号の表示例

9章

関数は新しい世界を開くか

前章まで紹介してきた機能は，実際のところBASICでも頑張れば実現可能なことで，Cの絶対的なメリットではありませんでした．それでは，これぞCの優れた点であると胸を張っていえる部分はどこにあるのか．これは筆者の個人的見解ですが，やはり関数を自由に作成して使えることではないかと思います．

関数はプログラムの構造をはっきりさせ，プログラミングのあいまいさをなくし，さらに関数を蓄積していくことによってプログラム開発の手間を大きく省くことができます．Cの関数には，自分でC言語の体系を拡張していける快感があります．関数を利用できるようになると，プログラミングの世界はぐんとひろがります．

関数の作成法とその使い方

プログラムを作る場合，それが大きなサイズのプログラムになればなるほど，全体を一度に書き上げることが難しくなります．そのため，どのようなプログラミング言語にも，プログラムをある限られた機能ごとに分け，まず各部分を完成させた上で全体を組み合わせる手段が用意されているものです．たとえばBASICではサブルーチン(GOSUB命令)が利用できました．

Cでは関数がその役割を担っています．といっても，数学の時間にサインコサインとかいって苦しめられた例のやつではなく，Cでは「ある機能を持った実行単位」のことを関数と呼びます(どちらも英語ではfunctionというのです)．

ここではCの最も簡単な部類に属する関数から，順に高度な機能のものまで，実際にいろいろな関数の例を見ていくことにします．

関数を定義する

関数を使うためには，それに先だって関数の定義を行う必要があります．関数の定義とは，その関数に実行させたい内容を指定して，さらに関数名をつける作業です．

ただしこれから述べていくように，渡されたパラメータを使うかどうか，あるいは戻り値を返すかどうかによって，関数定義の方法も少しずつ異なります．まずはもっとも簡単な，パラメータなし，戻り値なしの関数を例にとって，基本的な関数定義の方法を紹介しましょう．

●いちばん簡単な関数の定義

いちばん簡単なCの関数は，渡されるパラメータも持たず，戻り値も返さないものです．これは実行したい文をいくつかブロックにまとめて，それに

名前をつけただけの図 9.1 のような形式で定義されます.

```
                ┌─ここにセミコロンを付けないこと
                ↓
関数名( )
{
        実行したい文1 ;
        実行したい文2 ;
          :
}
```

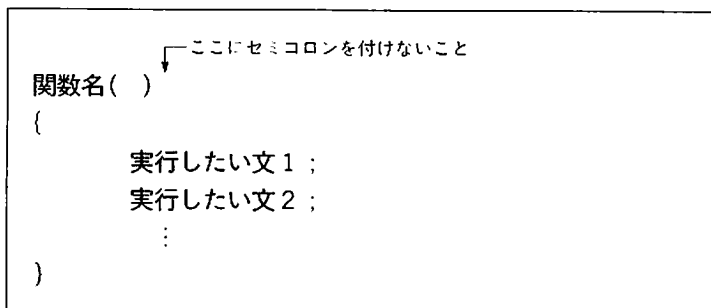



図 9.1　いちばん簡単な関数定義の形式

たとえば図 9.2 は, 文字を並べて箱を描くプログラムの例ですが, ここでは横に一本線を引く yoko() という関数と, 箱の両脇の縦線を書く tate() という 2 つの関数を定義して使っています.

```
#include <stdio.h>

yoko()
{
    puts("+-----+");
    putchar('¥n');
}

tate()
{
    puts("I     I");
    putchar('¥n');
}

main()
{
    int  i;

    yoko();                           ..............yoko( )を実行する
    for (i = 1; i <= 4; ++i)          ..............tate( )を 4 回繰り返す
        tate();
    yoko();                           ..............yoko( )を実行する
}
```

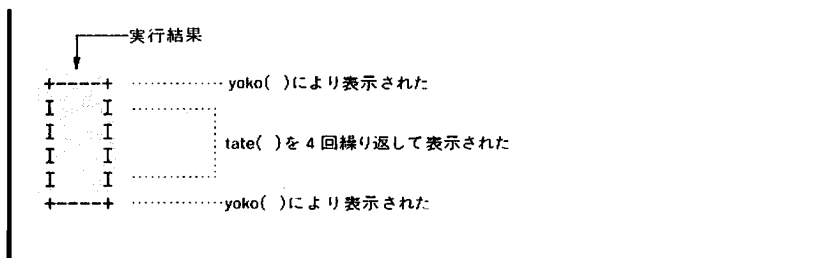



図 9.2　簡単な関数の定義とその使用例

いったん関数が定義されれば，それを実行するのはしごく簡単．たとえば上のプログラムの yoko() という関数ならば，その名前を使って「yoko()；」という関数呼び出し文を書くだけです．こうすると，yoko() の関数定義の中で指定したとおりに文が実行されるのです．

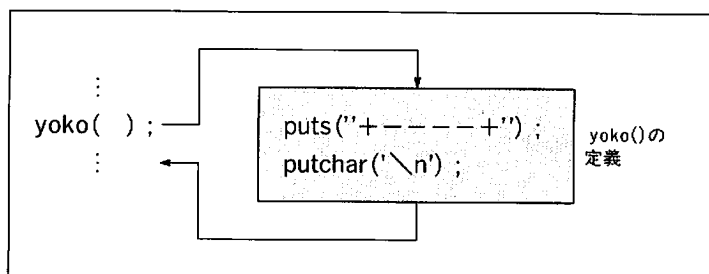



図 9.3　関数呼出と実行のしくみ

このように自分で定義した関数は，puts() や putchar() などのライブラリ関数とまったく同様に利用できます．if 文や for 文の中で使うことも自由です．図 9.2 のプログラムに次のような部分がありますが，この for 文では tate() という関数が 4 回繰り返して実行されるわけです．

```
for (i = 1; i <= 4; ++i)
    tate();
```

ここであげたような単純な関数は，ある作業をまとめて実行するだけですから，機能的にはBASICのサブルーチンとほとんど変わりません．ただしサブルーチンが単なる行番号で呼び出されていたのに比べると，Cの関数は意味のある関数名で呼び出せるぶん，プログラムが非常にわかりやすくなっています．

●関数定義はいつ行うか

こうして自分で定義した関数を利用して，さらに別の関数を定義することも可能です．ただし4章でも述べたように，関数を組み合わせて別の関数を作るときは，どの関数も使用する前にその存在がコンパイラにわかるようにしておかなくてはなりません．

一番簡単なのは，さきに関数定義を行い，そのあとで利用することですが，この順序をとらない場合(関数が定義される前にそれを利用する場合)は，とりあえず関数の使用宣言だけ行います．

本来この宣言は「関数の型」を考えにいれて行うべきなのですが，いま扱っているような単純な関数では，次の形式で宣言すればよいでしょう．

```
extern  関数名()；
```

4章では省略していましたが，正しくはexternを付けます．図9.4は，この関数宣言を使ったプログラムです．関数の宣言は2行目のように，すべての関数定義の外側で行ってもかまいませんし，12~13行目のように，その関数を呼び出したい関数定義の中で独自に宣言してもかまいません．

前者の場合，関数の使用宣言はプログラム全体を通して有効となり，以後は特別な宣言なしに自由にbox()という関数を使えるようになります．後者の場合，tate()とyoko()の2つの関数の宣言は，この宣言を行っているbox()の中だけで有効となり，ほかの関数がtate()やyoko()を利用したければ，それらが自分自身でまた宣言を行う必要があります．

```
Ø: #include <stdio.h>
1:
2: extern  box();
3:
4: main()
5: {
6:     box();
7:     box();
8: }
9:
1Ø: box()
11: {
12:     extern  tate();
13:     extern  yoko();
14:
15:     yoko();
16:     tate();
17:     tate();
18:     yoko();
19: }
2Ø:
21: yoko()
22: {
23:    puts("+----+");
24:    putchar('¥n');
25: }
26:
27: tate()
28: {
29:    puts("I    I");
3Ø:    putchar('¥n');
31: }
```

定義の中で宣言してもよい
(この場合，tate( )とyoko( )の関数宣言は，
box( )の中だけで有効となる)

図 9.4 関数宣言を行って，未定義の関数を利用する方法

なお，この本では面倒な手間を省くため，できるだけ関数宣言は行わずに関数定義の並べ方を工夫して対処していくことにします．

●関数は自分専用の変数を持てる

それぞれの関数は，その中で自分専用の変数を使うことができます．この変数は図 9.5 に示したように，関数本体のブロックの先頭(実行文が出て来る前)の位置で宣言します．

```
関数名( )
{
        変数の宣言 1 ;
        変数の宣言 2 ;   ←この位置で変数を宣言する
           :

        実行したい文 ;
           :
}
```

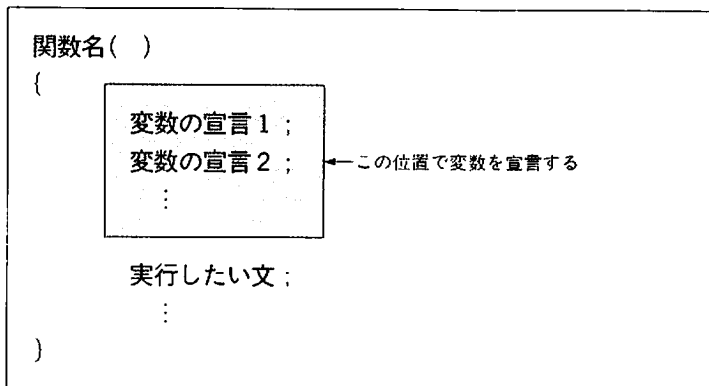



図 9.5 関数の中で変数を使いたい場合

図 9.6 は，アスタリスク「＊」を 7 文字×5 行の 4 角形に並べるプログラムです．ここではまずアスタリスクを 7 文字表示して改行する astro7() という関数を作り，それを 5 回繰り返して呼び出しています．

```
#include <stdio.h>

astro7()
{
    int  i; ……………この変数 i は main( )の変数 i とは無関係

    for (i = 1; i <= 7; ++i)
        putchar('*');
    putchar('¥n');
}

main()
{
    int  i; ……………この変数 i は astro7( )の変数 i とは無関係

    for (i = 1; i <= 5; ++i)
        astro7();
}

*******
*******
*******  ←――実行結果
*******
*******
```

図 9.6 それぞれの関数で使う変数は互いに無関係である

上のプログラムで,特に注意していただきたいのは,astro7()の関数定義の中でもiがループ変数として使われ,そのastro7()を呼び出すmain()の関数定義の中でも,やはりiがループ変数に使われていることです.

これがBASICならば,astro7()の実行中にiが書き換えられてしまうため,main()のループは意図したとおりには動いてくれないでしょう.しかし,Cではそうはなりません.なぜなら関数定義の中で宣言された変数は,その関数専用のものとみなされ,互いにまったく独立して扱うことができるからです.

つまり図9.6のプログラムでは,astro7()の中でいくら変数iを書き換えてもmain()の変数iには影響がありませんし,その逆にmain()の中でiの値を書き換えてもastro7()の変数iは少しも変更を受けないわけです.このように,ある関数の中だけで通用し,ほかとはまったく独立している変数を,**ローカル(局所的)な変数**と呼びます.

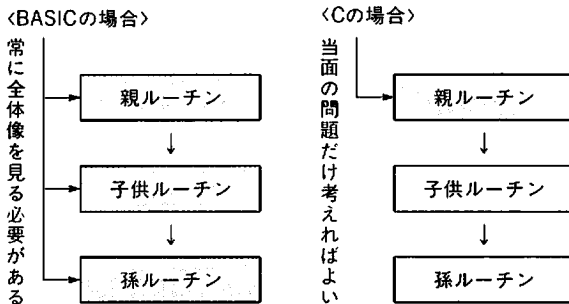



図9.7 ローカル変数の利点

このローカルな変数は,int型,char型など,どんなデータ型でも使えますし,配列変数でもかまいません.

Cでは,このローカル変数のおかげで,大規模なプログラムの開発がたいへん簡単にできるようになりました.BASICの場合は,どのサブルーチンでどんな変数が使われているかということをすべて心得ておかないと,予期しない変数の書き換えでいつプログラムが破綻するか知れません.

これに対してCでは,関数の中でどういう名前の変数が使われていよう

と，それを利用するほうはいっさい気にする必要がありません．関数というものは，その動作さえわかればOK. astro7()はアスタリスク記号を7個表示する，ただそれだけでよいのです．

●パラメータを持つ関数の作成

さて，図9.6のプログラムで作ったastro7()は，必ず7個のアスタリスクを表示する固定動作の関数でしたが，Cでは与えるパラメータによって，いろいろと動作を変えられるような関数も簡単に定義できます．

パラメータを持つ関数の定義形式は次のとおりです．

```
関数名(パラメータ変数名1 ; パラメータ変数名2………)
パラメータ変数の型宣言1 ;
パラメータ変数の型宣言2 ;
  ⋮
{
        ローカル変数の宣言
          ⋮
        実行したい文
          ⋮
}
```

図9.8　パラメータを持つ関数の定義の形式

パラメータの個数はいくつでも，どのデータ型でもかまいませんが，基本的に1つのパラメータが受けとれるデータは1つです．配列のようにまとまったデータをごそっと渡すことはできません(ただしこれには抜け道があり，10章で説明するポインタを利用すると，配列を受け取る関数を作ることが可能になります)．

図9.9に，astros()という関数の定義とその使用例を示しました．このastros()という関数は，カッコの中に適当な数値データを入れて，

```
astros(数値) ;
```

という形式で用いると，指定した個数ぶんのアスタリスクを表示します．この数値は16行目のように定数でも，19行目のように変数を使っても，あるいは22行目のように計算式で指定してもかまいません．

```
0: #include <stdio.h>
1:
2: astros(n)  ……………astros( )がnというパラメータを持つことの宣言
3: int  n;    ……………nがint型であることの宣言
4: {
5:     int  i;
6:
7:     for (i = 1; i <= n; ++i)  ……………1からnまで繰り返す
8:         putchar('*');
9:     putchar('¥n');
10: }
11:
12: main()
13: {
14:     int  i;
15:
16:     astros(5);  ……………パラメータに5を指定する
17:
18:     for (i = 1; i <= 4; ++i)
19:         astros(i);  ……………変数でも指定できる
20:
21:     for (i = 1; i <= 3; ++i)
22:         astros(4-i);  ……………計算式でも指定できる
23: }

実行結果
*****  ……………パラメータ5を与えてastros( )を実行した結果
*
**        1, 2, 3, 4とパラメータを変えながら
***       astros( )を実行した結果
****
***
**        3, 2, 1とパラメータを変えながら
*         astros( )を実行した結果
```

図9.9　表示するアスタリスクの数を変えられる関数

●パラメータ変数に関する注意

図9.10は，第1パラメータの文字c(char型)を，第2パラメータn(int型)の個数だけ表示するputline()という関数の定義です(これは4章の"TRIANGLE.C"というプログラムの中に出てきたものです).

このように，パラメータを受け取る変数(以後パラメータ変数と呼びます)は，それ以外の通常の変数と同様に，関数の定義の中で自由に値を参照したり変更することが可能です．

```
putline(ch, n)
char  ch;
int   n;
{
    int  i;

    for (i = 1; i <= n; ++i)
        putchar(ch);
}
```

図9.10　putline()の定義

関数を定義するときのパラメータ変数の型と，それを実際に利用するときに与えるデータの型は，かならず等しくなるように注意してください．特にMSX-Cでは関数の第1パラメータについては，char型とそれ以外のデータ型の区別が厳しく，これを間違えるとプログラムが思いどおりに動作してくれないこともあります．

たとえば上の putline()は，第1パラメータが char 型であることを前提に作られていますから，ここに char 型以外のデータを使ってはいけません．文字 A を5個表示させるのに，

```
putline('A', 5) ;              …… 文字定数は char 型
putline((char)65, 5) ;         …… キャストで char 型に合わせる
putline((char)0x41, 5) ;       ……         〃
```

などと書くことはできますが，

```
putline(65, 5) ;               …… 65 は int 型
putline(0x41, 5) ;             …… 0x41 は unsigned 型
```

と書いてしまうと正しく動作しないのです．

気をつけなくてはならないのは，パラメータを計算式で指定する場合のデータ型です．たとえば，次のような呼び出しを考えてください．

```
putline('A'+i, 10) ;           …… 変数 i は int 型とします
```

この場合，「'A'+i」という式は char 型＋int 型ですから，7章で説明したとおり，int 型になってしまいます．この式は char 型にキャストして，次のように書かなくてはなりません．

```
putline((char)('A'+i), 10) ;
```

●関数はプログラムをわかりやすくする

8章でカーソルを指定位置に移動させるエスケープシーケンスを紹介しました．これは，次のような文字コードを持つ4文字を順に出力することにより，カーソル移動を実現するものでした．

¥033, Y, Y 座標+32, X 座標+32

そこで，x と y の座標をパラメータに与えるとこのエスケープシーケンスを出力するような関数を作ってみます．関数の名前は BASIC の LOCATE 命令にならって locate()としましょう．図 9.11 に，locate()関数の定義とその使用例を示します．

```
#include <stdio.h>

locate(x, y)
int  x, y;
{
    printf("¥Ø33Y%c%c", y+32, x+32);
}

main()
{
    int  i;

    for (i = Ø; i <= 2Ø; ++i) {
        locate(i, i);
        printf("[%d,%d]", i, i);
    }
}
```

図 9.11　カーソル移動関数 locate()の定義と利用

さて，このプログラムを見てわかるとおり，locate()の定義の本体は，たった1行だけです．ですから，これを関数化したからといって，プログラムの手間を省くという点では，まったく役に立っていません．実際次のように書くほうがプログラムのサイズもずっと小さくなります．

```
#include <stdio.h>
main()
{
    int  i;

    for (i = Ø; i <= 2Ø; ++i) {
        printf("¥Ø33Y%c%c", i, i); ·············1行で書けるが意味がわかりにくい
        printf("[%d,%d]", i, i);
    }
}
```

図 9.12　locate()を定義しないで書いたプログラム

しかし，図9.11のように，あえてlocate()という関数を作ることには，プログラムの手間を省くという以上の大きなメリットがあるのです．それはプ

ログラムのわかりやすさです．つまり，

```
printf("¥033Y%c%c", i, i);
```

と書いても，これがカーソル移動を意味するとは，なかなか理解できません．それに対して，

```
locate(i, i);
```

と書かれていれば，誰が見てもこれはカーソルを(i, i)の位置に移動させるものだと判断できます．大規模なプログラムを作るときには，このようなプログラムの理解しやすさという要素も非常に大事です．その意味からも関数はどんどん利用すべきなのです．

●関数の実行を途中で終わらせる

関数は，通常はその定義の最後の文まで実行すると終わりになりますが，それ以外に次の return 文で強制的に終了させることもできます．

```
return;
```

この return 文を使った例として，さきほどの locate()を，座標が(0,0)～(39,19)の範囲外であればカーソル移動を行わずに即座に終了させるように改良してみましょう．この定義は図 9.13 のようになります．

この return 文は，いつでもどこでも実行可能です．ループが何重にも入れ子になっているような状態の奥底からでも，いっきに関数を終了させることができます．

```
locate(x, y)
int  x, y;
{
    if (x < 0 || x > 39 || y < 0 || y > 19) ……画面からハミ出した時は...
        return;                             ……戻る

    printf("¥033Y%c%c", y+32, x+32);
}
```

図 9.13 return 文は関数を中断できる

戻り値を返す関数

5章でも少し触れましたが，Cの関数には，なんらかのアクションを目的として使うputchar()のようなものと，戻り値を利用するために使うgetch()のようなものの2種類があります．

前者については，すでに前の節で述べました．ここでは戻り値を返す関数の宣言とその使用法を説明します．

●戻り値を持つ関数の宣言

戻り値を持つ関数を宣言するには，図9.14のように関数名の前に戻り値のデータ型を指定し，戻り値指定付きのreturn文で戻り値を返します．

```
戻り値の型　関数名(　)
{
        実行したい文
           ⋮
        return　戻り値;
}
```

図9.14　戻り値を持つ関数の定義

戻り値を持つ関数の例を図9.15に示します．ここで定義しているadd()は渡された2つのパラメータの和を返す関数です．

このような関数の戻り値は，変数に代入したり，計算式の中の1つのデータとして利用することができます．たとえば，

```
i = add(1, 2);          …… iに(1+2)が代入される
```

と書けば，add()は1+2を計算して3という戻り値を返し，それが変数iに代入されます．次のように，関数に渡すパラメータの中でさらに関数呼び出しを書くこともできます．

i = add(1, add(2, 3)) ; …… iに(1+(2+3))が代入される

```
#include <stdio.h>

int  one()
{
    return  1; ……………つねに1を返す
}

int  add(x, y)
int  x, y;
{
    return  (x + y); ……………2つの数値の和を返す
}

main()
{
    printf("1 + 1 == %d¥n", add(1, one());
}
```

図9.15　戻り値を持つ関数の例

●戻り値のデータ型

関数の戻り値のデータ型は，その宣言で指定した型に一致します．これは int 型でも char 型でも unsigned 型でもかまいませんが，戻り値として返せるデータはたった1つです(ただし10章で説明するポインタを利用すると，複数の戻り値を返す関数を作ることも可能になります)．

return 文で返すデータの型は，自動的に指定した関数のデータ型に合わされます．たとえば図9.16を見てください．これは，与えられたパラメータに n を加えたコードの文字を返す関数 next()の定義です．

```
char next(c, n)
char c;
int  n;
{
    return (c + n);
}
```

図9.16　戻り値のデータ型は関数宣言が決める

この関数では，return 文に指定された戻り値は char 型＋int 型で，int 型のデータになってしまうように思えます．しかしこの関数は char 型を返すと宣言されているため，結果は自動的に char 型に変換されるのです．つまり，この next() という関数の戻り値は，かならず char 型であることが保障されています．たとえば次のように，putchar() のパラメータに渡しても問題はありません．

putchar(next('A'))；　……　文字Bが表示される

● VOID 型について

かつては戻り値を持たない関数は「型がない関数」として扱われていましたが，最近では，戻り値を持たない関数を void 型という特殊な型名で呼び，形式上どんな関数にも戻り値のデータ型がある，とすることがあります．ただし MSX-C では，これを大文字の VOID 型と呼んでいます．

VOID 型の指定は，ほかのデータ型と同様に関数名の前に置きます．たとえば，図 9.17 は画面をクリアする cls() という関数の定義です．

```
VOID  cls()
{
    putchar('¥f');
}
```

図 9.17　戻り値を返さない関数は VOID 型と宣言してもよい

戻り値を持たない関数に対して VOID の宣言を行っても行わなくても，関数の動作には変わりありません．しかし，これによって戻り値を返さない関数であることが強調され，プログラムの意味がわかりやすくなるという効果があります．MSX-C のサンプルプログラムなどを見ても，戻り値を持たない関数は，かならずこの VOID 型が宣言されています．

記憶クラスとスコープ

関数のなかでローカルな変数が利用できることはすでに説明しましたが，ときには複数の関数間で同じ変数を共有したい場合もあります．このようなときは，変数が関数ごとに独立していては困ります．またローカルな変数といっても，一時的に作業用として使えればよい場合もあれば，関数を呼び出すごとに前回使ったデータをまた参照したい場合もあります．

ここでは，これらの要求に応える変数の宣言の方法と，その具体的な使用法について説明しましょう．

変数の記憶クラス

Cの変数は，それぞれ記憶クラスと呼ばれる性質を持っています．記憶クラスは変数の寿命を決定します．変数の寿命とは，関数を呼び出すごとに新しい変数が用意されるのか，あるいはプログラムの実行開始時から終了時まで常に同じ変数が参照されるのかどうかを意味します．

MSX-Cの変数には，寿命の違いによって，**表9.1**に示す2つの種類の記憶クラスがあります．

記憶クラス	変数の寿命
auto変数	関数を呼び出すごとに新しく用意される
static変数	プログラム全体を通して同じ変数が使われる

表9.1　変数の記憶クラス

●その場かぎりのauto変数

ここまで使ってきた変数は，すべてauto変数と呼ばれる記憶クラスに属するものでした．このauto変数は，関数が呼び出されるごとにあるメモリ領

域に一時的に用意され，その関数の実行が終了すると捨てられてしまいます．

関数内で宣言した変数は，黙っていれば自動的にauto変数とみなされますが，auto変数であるということを強調したいときには，次のような宣言を行うこともできます．

```
auto int  i ;
```

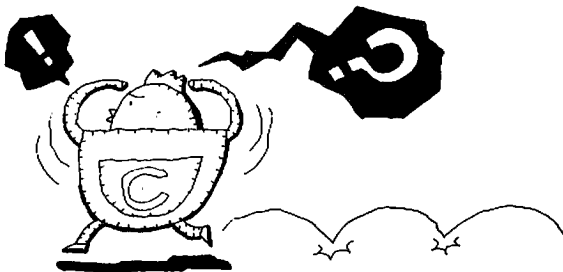
もっとも，このautoは，あってもなくてもプログラムの動作はまったく変わらないため，いちいち宣言されることはまずありません．普通は省略してしまいます．

● いつまでも覚えている static 変数

通常のプログラムで使う変数は，auto変数であっても何も問題はないのですが，用途によっては，関数が終了してもその中で使った変数の値を消したくない場合があります．

このような場合には，static変数を使用します．static変数は，次のようにstaticというキーワードを頭に付けて宣言します．

```
static int  i ;
```

図9.18のプログラムは，このstatic変数を利用して作った乱数関数rnd()の例です．rnd()の中にはrというstatic変数が用意されていて，呼び出されるごとに「r = r * 257 + 1 ;」という計算が行われます．このように前回の値を覚えておいてくれるのは，rがstatic変数であるからです．

```
#include <stdio.h>

int  rnd()
{
    static int  r;

    r = r * 257 + 1;
    return  r;
}

main()
{
    int  i;

    for (i = 1; i <= 100; ++i)
        printf("%10d", rnd());
}
```

図 9.18 static 変数の使用例

なお，運がよければ，auto 変数でも次の関数呼び出しまで値が捨てられずに残っていることがありますが，それは本当に運がよかっただけ．プログラムはなによりも確実さが命です．幸運に頼ってはいけません．

変数のスコープ

ここまで見てきた C の変数は，すべて 1 つの関数の中だけでしか参照できないローカルな変数でした．これに対して，BASIC で使う変数のように，サブルーチンの中でもメインルーチンの中でも自由に参照できる変数をグローバルな変数と呼びます．

このように，ある変数が通用する範囲を変数のスコープといいます．C の変数には表 9.2 に示した 2 種類のスコープがあります．

スコープ	変数の宣言方法	スコープの範囲
ローカル変数	関数の中で宣言する	1 つの関数
グローバル変数	関数の外で宣言する	1 つのプログラム

表 9.2 変数のスコープ

●関数の中だけで通用するローカル変数

ここまで述べてきたとおり，関数の中で宣言した変数は，その関数の中だけで通用するローカル変数となります．

●どこでも通用するグローバル変数

ローカル変数に対して，図9.19のようにすべての関数定義の外側で宣言した変数はグローバル変数と呼ばれ，プログラム全体から参照できる変数となります．ただしグローバル変数といえども，宣言の前に使用することはできません(そのため，グローバル変数はたいていプログラムの先頭でまとめて宣言されます)．

```
#include <stdio.h>

int  Sum;
int  Product;

addmul(x, y)
int  x, y;
{
    Sum = x + y;
    Product = x * y;
}

main()
{
    int  i, j, k;

    for (i = 1; i <= 9; ++i) {
        for (j = 1; j <= 9; ++j) {
            addmul(i, j);
            printf("%d + %d == %3d, ", i, j, Sum);
            printf("%d * %d == %3d¥n", i, j, Product);
        }
    }
}
```


図9.19 グローバル変数を使ったプログラム

グローバル変数を使うと，関数の間で複数のデータをやりとりすることが可能です．ここで定義しているaddmul()という関数は，2つの数値x，yの和と積を求めるものですが，この答を両方とも関数の戻り値として返すことはできません．関数の戻り値として返せるデータはたった1つです．

そこで，ここは2つの答をそれぞれSumとProductというグローバル変数に代入することにします．addmul()を呼び出したmain()のほうでは，関数の戻り値として答を受け取るのではなく，グローバル変数のSumとProductを見て答を知ればよいわけです．

このように，グローバル変数は使い方によってはたいへん便利なのですが，BASICの変数と同様に，どこで値が書き換えられるかわからない欠点も持っています．グローバル変数をやたらに書き換えると，プログラムの動作が非常に理解しにくくなってしまいます．

そこで，この本では，グローバル変数はちょっと特別な変数なのだぞ，ということを強調するために，グローバル変数の名前は先頭の文字をアルファベットの大文字にするという規則を設けることにしました．

10章

ポインタと配列と文字列

マシン語でプログラミングした経験のある方はご存じだと思いますが，マシン語では，どのアドレスにどういう形でデータを記録するかをプログラマが自分で管理しなければなりません．しかしこのアドレスというやつは根が単なる数字ですから，いちいち覚えておくのも面倒ですし，間違えずに使いこなすのがなかなか大変です．

そこでCでは，アドレスをポインタという特別なデータと考えて，その扱い方をきちんと定めることにしました．おかげでCではアドレス操作が非常に簡単で便利なものとなっています．またポインタは，配列や文字列とも分かちがたい関係をもっています．

本章では，このポインタの利用法を紹介します．

10 1 ポインタを使ったプログラム

ポインタとは「あるものを指し示す存在」です．たとえばポインタという種類の狩猟犬は，獲物のいる場所を前足で指し示し，『あそこですよ御主人さま』と教えてくれるのだそうです(真偽はサダカでない)．Cでは「データを指し示すもの」をポインタと呼びます．単純なアイデアですが，これがあるおかげで，Cは大変に柔軟性を持つプログラミング言語となりました．

アドレス演算子「&」と間接演算子「*」

コンピュータの中では，データはメモリと呼ばれる記憶場所に蓄えられます．MSXの場合，メモリは65536個の小さな部屋に分割され，それぞれ1バイトのデータを記憶させることができます．そして，その中から特定の場所を指定するために，メモリには0番地～65535番地(0x0000～0xffff)の一連のアドレスが順に振られています．

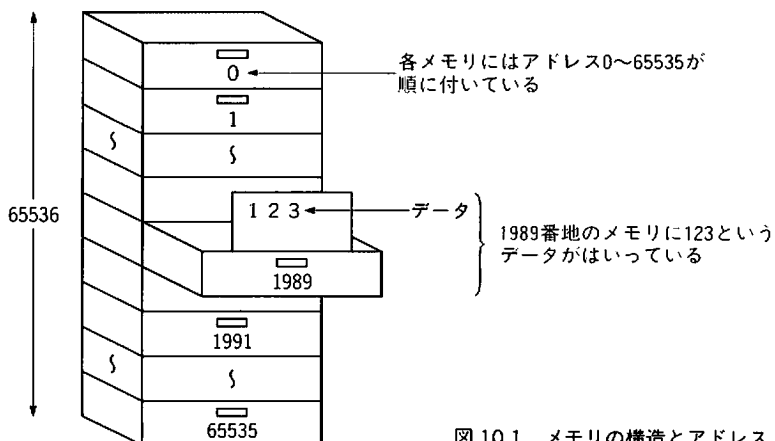



図 10.1　メモリの構造とアドレス

プログラムの中で使う変数や配列は，すべてこのメモリ上に割り当てられています．BASIC の場合は，かなり特殊なプログラムでもない限り，この変数のアドレスを知る必要はありませんでした．それに対して C では，これから述べるように，変数のアドレスを利用するといろいろ面白いプログラムを作ることが可能です．

●変数のアドレスを調べる

変数のアドレスを知るためには，&記号で表される**アドレス演算子**というものを使います．この&記号を単独で変数の前に付けると，その変数のメモリ上のアドレスが求められます．たとえば，

&i

は，変数 i のアドレスを表します．また&演算子は普通の変数だけではなく，配列変数の要素に対して用いることもできますから，

&a [i]

と書けば，配列要素 a [i] のアドレスが得られます．これらを実際にプログラムの中で使った例を図 10.2 に示しましょう．

```
#include <stdio.h>
main()
{
    int  i;
    int  a[3];

    printf("i ﾉ ｱﾄﾞﾚｽ ﾊ %u¥n", &i);
    for (i = Ø; i <= 2; ++i)
        printf("a[%d] ﾉ ｱﾄﾞﾚｽ ﾊ %u¥n", i, &a[i]);
}
```

実行結果

```
i ﾉ ｱﾄﾞﾚｽ ﾊ 54772
a[Ø] ﾉ ｱﾄﾞﾚｽ ﾊ 54774
a[1] ﾉ ｱﾄﾞﾚｽ ﾊ 54776
a[2] ﾉ ｱﾄﾞﾚｽ ﾊ 54778
```

MSXのハードウェア構成により，表示されるアドレスはこれと異なる場合もあります

図 10.2　&演算子で変数のアドレスを求める

●変数のアドレスに数値を加えるとどうなるか

このように&演算子を使って求められる変数のアドレスは，通常の数値データとは，ちょっと違う性質を持っています．試しに図 10.3 のプログラムを実行してみてください．ここでは int 型変数 i のアドレスと，それに 1 を加えた値，そして 2 を加えた値を表示させています．

```
#include <stdio.h>
main()
{
    int  i;

    printf("&i     == %u¥n", &i);
    printf("&i + 1 == %u¥n", &i + 1); ··············&iに1を加えた値を表示
    printf("&i + 2 == %u¥n", &i + 2); ··············&iに2を加えた値を表示
}
```

実行結果

```
&i     == 54778
&i + 1 == 54780 ··············&iの値より2だけ大きい
&i + 2 == 54782 ··············&iの値より4だけ大きい
```

図 10.3　アドレスに数値を加える

すると図 10.3 に示すとおり，&i に 1 を加えるとアドレスは 2 増加し，&i に 2 を加えるとアドレスは 4 だけ大きくなりました．つまり，加えようとした数値の 2 倍の数がアドレスに足されているのです．

●ポインタ型のデータとは

図 10.3 のような結果が得られるのは，コンパイラが&i をポインタ型のデータとみなすからです．

ポインタ型は，アドレスを専門に扱うために用意されたデータ型です．これは通常の数値データと違い，掛け算や割り算の対象にはできません．できるのは，次のように，そのアドレスを基準としていくつか先のアドレスを求めること(ポインタの足し算)，あるいはいくつか前のアドレスを求めること(ポインタの引き算)だけです．

&i　＋　n　(&iのn個先の変数のアドレスを求める)
&i　－　n　(&iのn個前の変数のアドレスを求める)

ところで，n個だけ先や前にある変数といっても，どのようなデータ型の変数を考えるかで結果は異なってきます．これは変数のデータ型に応じて，それがメモリ上で占めるサイズが異なるためです．

たとえば，MSX-Cではint型の変数は2バイトのメモリを使い，char型の変数は1バイトのメモリを使います．ですから図10.4のように，1つ先の変数といってもint型の場合なら2バイト先，char型ならば1バイト先ということになります．さきほどの図10.3のプログラムで&iに1を加えるとアドレスが2つ増えたのも，&iがint型の変数のアドレスだったからなのです．

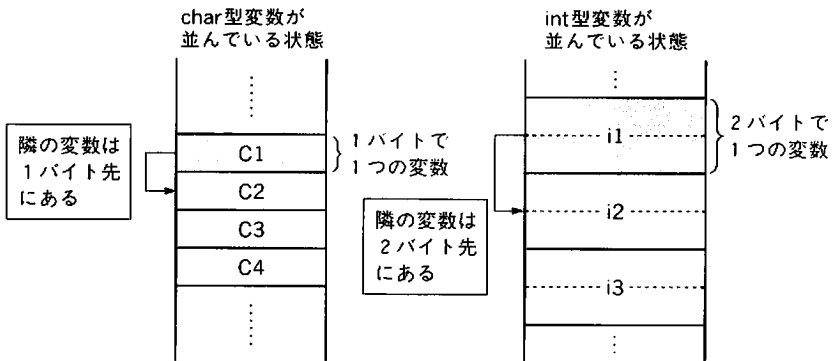



図10.4　「1つ先の変数」はどこにあるか

そのためポインタを扱うときは，それがint型変数のアドレスなのか，はたまたchar型変数のアドレスなのか，つねにそのアドレスに位置する変数の型まで区別して考えなければなりません．

int型変数のアドレスの性質を持つポインタを，Cではint型へのポインタと呼びます．同様に，char型へのポインタといえば，これはchar型変数のアドレスに相当します．

●ポインタを変数に代入して使う

ポインタ型のデータは，同じポインタ型の変数に代入することができます．ポインタ型の変数は，次のように変数名の前にアスタリスク記号「*」を置いて宣言します．

```
データ型名　　*変数名；
```

たとえば int 型へのポインタを格納するために p という名前の変数を使おうと思ったら，

```
int    *p;
```

という宣言を書くことになります．

このようなポインタ型の変数も，やはりポインタ型データの1つですから，

```
p = p + 1;
```

という足し算は，p に 1 を加えるのではなく，「**p を 1 つ先のアドレスに進める**」という意味になります．つまり，この代入文によって p の指し示すアドレスは 2 だけ増加するわけです．これは，

```
++p;
```

と書いても同様です．ポインタに対する加減算はすべて，その指し示すデータのサイズ単位でポインタを前後にいくつか移動させることになるのです．

●ポインタの指し示すデータを参照する

ポインタ型データの頭に単独の「*」を付けたものは，そのポインタの指し示すメモリの内容を表します．たとえば，

```
*p
```

というのは「p が指し示すアドレスにあるメモリの内容」を意味します．ここでは p が int 型へのポインタとして宣言されていますから，*p は int 型のデータとして扱われます．

この*p は，次のように普通の int 型の変数を使うのとまったく同じ感覚

で，値の代入や参照ができます．

```
*p  =  12345;        …… *pへの代入
 i  =  *p;           …… *pの参照
```

このときのメモリの状態を図で示すと，図10.5のようになります．つまり＊pの値は，「『変数pの内容』で示されるメモリの内容」という，2段階の手間を通して間接的に参照されているのです．このことから，＊演算子を**間接演算子**とも呼びます．

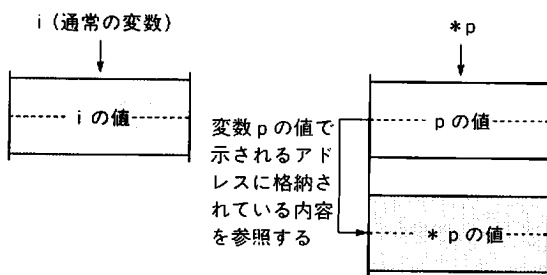



図10.5　＊pが参照される仕組み

配列とポインタの関係

前項でポインタの基本的な扱い方を紹介しましたが，これだけの説明では，いったいポインタのどこが便利なのか，なぜポインタを使う必要があるのか，いまひとつ理解できなかったと思います．

実はポインタというものは，Cのほかの機能と組み合わせて使うことによってはじめて大きな利用価値が生まれるのです．ここではまず，配列とポインタを組み合わせた場合を見てみましょう．

●ポインタを使って配列を扱う

ポインタに1を足したものは，それがint型へのポインタであってもchar型へのポインタであっても，つねにその1つ先の変数を指すということは，

すでに述べたとおりです.

これが普通の変数ならば，その変数自身と隣にある変数の間にはなんの関係もありませんから,「1つ先の変数」の内容などを調べる意味はありません.

しかし配列を扱う場合になると，話が違ってきます．配列とは，そもそもint型ならint型のデータを順序よく並べたものですから，a [i] の1つ先のアドレスには，かならずa [i+1] が位置しています．そのためポインタを使うと，配列をうまく扱うことが可能となります.

たとえば図10.6は配列a[3]の内容をすべて表示する簡単なプログラムですが，ここではポインタ変数pの値を1つずつ先に進めながら，配列要素を順に表示しています.

```
#include <stdio.h>
main()
{
    int  a[3];
    int  *p;
    int  i;

    a[Ø] = 1957;  ┐
    a[1] = 7;     ├ 配列の中身を準備する
    a[2] = 21;    ┘

    p = &a[Ø]; ……………pに配列の先頭アドレスを代入
    for (i = Ø; i <= 2; ++i) {
        printf("%d " *p); ……………pの指すデータを表示する
        ++p;              ……………pを1つ先の要素に進める
    }
}


1957 7 21 ←――――実行結果
```

図10.6　ポインタを使って配列を参照する

この図10.6では，ループ変数iを使ってforループを実現しましたが，あるいはもっと直接に，p自身をループ変数としてしまうことも可能です．その場合は次のようなfor文になります.

```
for (p = &a [0]; p <= &a [2]; ++p)
    printf("%d", *p);
```

はじめのうちは，ポインタを使ったプログラムはわかりにくいと思うのですが，ポインタの扱いに慣れてくると，かえってこのように書いたほうがプログラムがすっきりして見えてくるから不思議です．

●配列名の正体はポインタである

図10.6のプログラムでは，配列a [] の先頭へのポインタを，

```
&a [0]
```

という形で求めていましたが，これは次のように書くこともできます．

```
a
```

aというのは，ただ配列名をそのまま書いただけじゃないかって？　そうなのです．実はCでは，**配列名はその配列の先頭要素へのポインタでもある**のです．ためしに図10.7のプログラムを実行してみてください．&a [0] とaでは，まったく同じ値が表示されるはずです．

```
#include <stdio.h>
main()
{
    int  a[3];

    printf("&a[0] == %u¥n", &a[0]);
    printf("a     == %u¥n", a);
}
```

図10.7　&a [0] とaは同じ値になる

また，配列名aが配列の先頭要素a [0] へのポインタなのですから，a+1はその次の要素のa [1] を指すポインタ，つまり&a [1] に等しいわけです．同様に考えると，a+iという計算式は，&a [i] とまったく同じアドレスを指

すポインタであることもわかるでしょう．このことは，次に述べる関数とポインタの組み合わせで威力を発揮します．

関数定義にポインタを利用する

このポインタを関数定義と組み合わせると，配列をパラメータとして渡したり，関数から複数個のデータを受け取るなど，いままでの説明にはなかった機能を持つ関数が実現できます．

●ポインタを関数に渡すには

まず基本的な事実は，ポインタは関数のパラメータとして使うことができるということです．たとえば図 10.8 に示すのは，int 型へのポインタをパラメータとする関数 clear()の定義例です．この関数は，与えられたポインタの指すメモリに 0 を書き込みます．

```
#include <stdio.h>

clear(p)  ……………clear( )は p をパラメータとする
int   *p; ……………pは int 型へのポインタである
{
    *p = 0; ……………pの指すアドレスをゼロクリアする
}

main()
{
    int  i;

    clear(&i); ……………i をゼロクリアする
    printf("i == %d", i);
}
```

図 10.8 ポインタをパラメータとする関数

このとき，main()の中で，

clear(&i)；

を実行すると，図 10.9 のような操作が行われて，変数 i に 0 が代入されます．

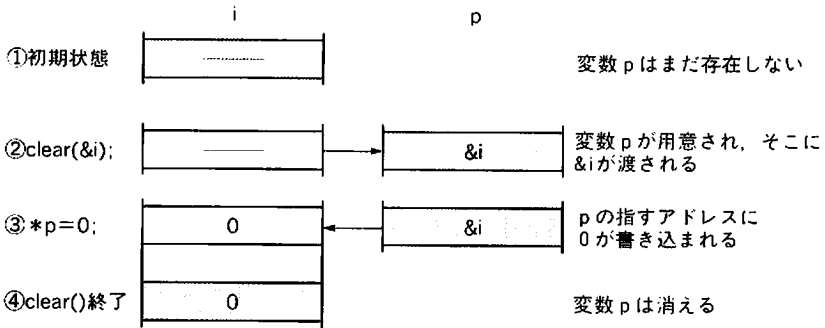



図 10.9　clear() の動作の仕組み

ここで注目して欲しいのは，clear() の実行によって，main() の使っている**ローカル**変数 i が 0 になることです．

関数の中で宣言したローカル変数は，その関数の中だけでしか参照できないというのがタテマエですから，通常は関数がどんな操作を行おうとも，呼び出し側の変数は変更を受けません．しかし，このようにポインタを渡すことによって，呼び出し側の変数に影響を及ぼす関数が作れるのです．

●関数から複数の値を受け取る方法

上で述べた「**ポインタ渡し**」のテクニックを使うと，1 つの関数で，1 度に複数の答を求めることができます．図 10.10 はその実例で，2 つの数値の和と差を計算する addsub() という関数の定義です．

```
addsub(x, y, s, d)
int  x, y;  ················xとyはint型
int  *s, *d; ···············sとdはint型へのポインタ
{
     *s = x + y; ···············sの指すアドレスにxとyの和を書き込む
     *d = x - y; ···············dの指すアドレスにxとyの差を書き込む
}
```

図 10.10　1 度に複数の答を計算する関数

このaddsub()を使うときは，初めの2つのパラメータに計算したい数値データ，後の2つのパラメータに答を受け取る変数へのポインタを渡します．たとえば，1000と700という2数の和と差を，sumおよびdifという変数に代入するには，

```
addsub(1000, 700, &sum, &dif) ;
```

という形でこの関数を実行します．関数の実行後にsumとdifの内容を調べてみれば，それぞれ1700と300が代入されているはずです．

●配列を扱う関数の作り方

またポインタを利用すると，配列を関数のパラメータとして渡すことができるようになります．まず図10.11を見てください．これは，渡されたポインタ以降に存在するn個ぶんのint型データを表示するための，nprint()という関数の定義です．

```
nprint(p, n)
int  *p;
int  n;
{
    while (--n >= 0) {              ……………ループをn回繰り返す
        printf("%d ", *p);          ……………＊pの値を表示し，
        ++p;                        ……………pを1つ先に進める
    }
}
```

図10.11　配列の内容を表示する関数

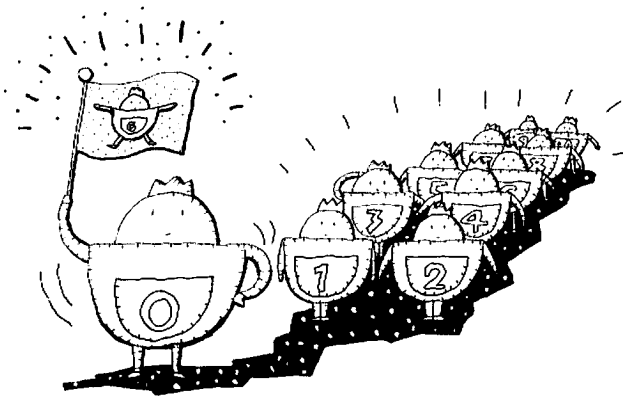
このnprint()を使って，たとえば5個の要素を持つ配列a[]の内容をすべて表示させるには，

```
nprint(a, 5) ;
```

と書けばよいのです．さきほど述べたように，配列名aはその先頭要素へのポインタですから，これで配列の先頭から5個ぶんの連続したデータが表示されるというわけです．

なお，aの代わりに&a［0］を使って，

```
nprint(&a [0] , 5) ;
```

と書いても同じ動作になりますが，&a［0］という書き方が「配列の先頭要素のアドレス」という意味あいを持つのに対して，aの場合は「配列全体を代表する値」のニュアンスが強くなります．これはもう主観の問題なのですが，このように直感的にわかりやすい表記が選べることも，Cの良さの1つといってよいでしょう．

10-2 文字列の操作

さて，前節でポインタの動作と利用法をつらつらと眺めてきましたが，これらの機能をたいへんにうまく使っているのが，実はCの文字列です．Cの文字列はchar型データの配列で表されるため，ポインタを利用すると非常に巧妙な操作が可能です．ポインタの扱い方を覚えるには，文字列操作を例にとるのが一番かもしれません．

文字列の正体を調べる

ここまでは，単に文字を並べてダブルクォートで囲んだものが文字列であるという説明をしてきましたが，文字列に対していろいろな操作を行う場合は，そのような外見の問題だけではなく，文字列がどのように構成されているかも知る必要があります．

そこで，文字列操作の方法を紹介する前に，まず文字列の構造について説明しておくことにしましょう．

●文字列の構造

Cの文字列は，基本的にはchar型データの配列の形をとっています．ただし文字列の最後にはかならずヌル文字(文字コード0に相当する文字：'¥0')が付け足され，実際の文字数より1バイトだけ大きなサイズのメモリがとられます．

たとえば図10.12に示すように，"MSX"という文字列をプログラムで使用すると，メモリの中では4バイトの領域が確保され，その先頭から順に，M，S，X，の文字コードが格納されます．そして最後には自動的にヌル文字が追加されます．

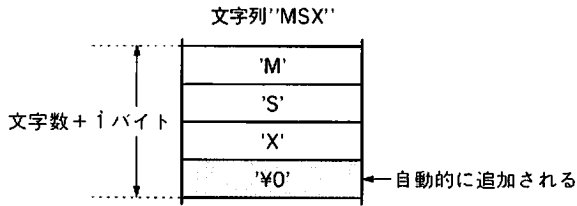



図 10.12　文字列の内部構造

●文字列はそれ自身へのポインタである

文字列はこのように char 型の配列の形をとっているため，前節で述べた配列とポインタの関係を利用すると，さまざまな操作を加えることが可能になります．

ただ，ここで1つ問題があります．普通の配列，たとえば a [3] という配列ならば，&a[0]あるいは a と書くことによって，そのメモリ上のアドレスを知ることができました．ところが文字列には特定の配列名というものがないため，同様の方法を使ってアドレスを求められないのです．

そこで，C 言語を開発した人々は，たいへんにうまいことを考え出しました．文字列自身をその文字列へのポインタとして扱おうというのです．

たとえば，〝MSX〟という文字列は，我々にとっては文字列以外のなにものにも見えませんが，コンパイラはこれを〝MSX〟という文字列の先頭へのポインタと解釈するのです．

したがって，ポインタ型の変数 p を使って，

```
p = "MSX" ;
```

と書けば，〝MSX〟という文字列の先頭アドレスが p に代入されることになりますし，いままで puts()で文字列を表示するのに，ごく当然のように，

```
puts("MSX") ;
```

などと書いてきたのも，コンパイラにとっては「〝MSX〟という文字列の先頭アドレスを puts()に渡す」という意味だったわけです．

文字列操作の実例

ここまでの話をまとめると，文字列には次の3つの重要な性質があることがわかります．

① 文字列はchar型データの配列である

② 配列の最後にはかならずヌル文字'¥0'が付いている

③ 文字列はそれ自身へのポインタと解釈される

文字列を扱う場合は，これらの性質をうまく利用していくことになります．以下に実際の関数定義の例を2つあげてみますので，これを参考に，文字列とポインタを組み合わせたプログラミングの感覚をつかんでください．

●文字列を表示する関数を作る

まず，関数のパラメータに文字列を使う例として，渡された文字列をそのまま表示する，つまりputs()と同じことを行う関数を作りましょう．図10.13がそのプログラムです．

```
newputs(p)
char  *p;
{
    while (*p != '¥0') { ·············· *pが文字列の終わりでない間
        putchar(*p);     ·············· *pを表示して
        ++p;             ·············· pを1進める
    }
}
```

図10.13 文字列を表示する関数

この関数は，たとえば，

newputs("MSX")；

という形で呼び出します．さきほど述べたとおり，このときnewputs()のパ

ラメータpには，"MSX"という文字列の先頭へのポインタが渡されます．あとは最後の文字まで，つまりヌル文字'¥0'が出て来る直前まで，pを進めながら1文字ずつ表示していけばよいわけです．

● 文字列コピーの関数を作る

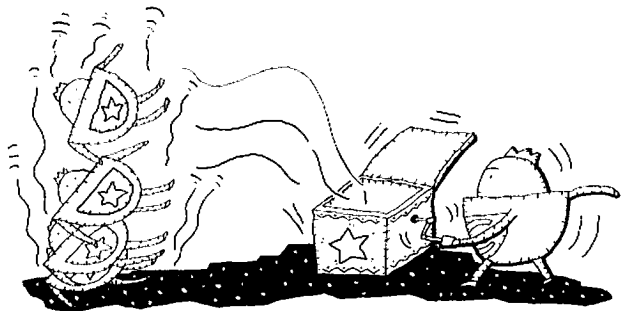
さて，次は文字列を別の場所にコピーする関数の定義例です．図10.14にプログラムを示します．

```
copy(p, q) ……………文字列qをpの位置にコピーする
char  *p, *q;
{
    while (*q != '¥Ø') { ……………文字列の最後までループする
        *p = *q; ……………1文字コピー
        ++p; ……………pを1つ先のアドレスに進める
        ++q; ……………qを1つ先のアドレスに進める
    }
    *p = '¥Ø'; ……………文字列の最後の印を付ける
}
```

図10.14　文字列をコピーする関数

この関数を実行するには，コピーを入れる入れ物と，元データになる文字列を指定する必要があります．たとえば，char型の配列b[10]に，"MSX"という文字列をコピーするならば，

```
copy(b, "MSX");
```

と書きます．

10 3 構造化データを使う

前の節では，主に文字列を使った単純なデータの入出力について述べました．ここでは，もう少しまとまったデータを扱う方法を紹介しましょう．データベースなどを扱う際に便利な構造化データの使い方です．

構造体の使い方

BASICでは，複数のデータを関連付けて扱う手段は配列だけでした．Cではこれに加えて**構造体**が利用できます．配列は基本的に同じ型のデータをいくつも並べたものですが，構造体は異なる型のデータをまとめて扱うことが可能です．

●構造体の定義

たとえば名簿を作ることを考えましょう．この名簿には各人の名前と性別と年齢を記載するものとして，次のような3項目を用意することにします．

```
char  name [32] ;        ……  名前文字列
char  sex ;              ……  'M'なら男性，'F'なら女性とする
int   age ;              ……  年齢
```

これらの項目は，3つまとまって1人の人間を表すデータになるわけです．Cではこのようなデータの集まりに，「人間型」とでもいうべき名前を付けて，新しいデータの型を作ることができます．

図10.15に，実際にその新しいデータの型を作る方法を示します．

この宣言によって，char型配列name [32]，char型変数sex，int型変数ageの3者からなる複合データ型が定義されました．

```
struct person {
    char name[32];
    char sex;
    int  age;
};
```

図 10.15　struct person 型の宣言

このようにいくつかのデータを組み合わせて作り出したデータを，構造体と呼びます．また，構造体を構成する各要素(ここでは，name [32]，sex，age)を，構造体のメンバーと呼びます．

構造体は「struct」の後ろに特有の呼び名を付けて区別します．上の例の構造体ならば，struct person 型ということになります．

実際にこの構造体を使うには，まず struct person 型の変数を用意しなくてはなりません．たとえば，

```
struct person   taro ;
```

という変数宣言を行うと，struct person 型の taro という変数が利用できるようになります．また，

```
struct person   family [3] ;
```

という宣言を行えば，これは struct person 型の要素 3 個からなる family [] という配列になります．

●構造体の参照と値の代入

構造体を扱う場合は，それぞれのメンバーごとに個別に操作を行う必要があります．ある構造体変数の中の特定のメンバーを指定するには，その構造体変数名の後ろに，ピリオドとメンバー名を続けます．たとえば，

```
taro.age
```

と書くと，「taro の age」が指定されることになります．

さきほど struct person 型を定義するときに，メンバーage は int 型と宣言

しましたから，この taro.age も int 型の変数と同様に扱えます．taro の年齢を 7 才としたければ，次のような代入文を実行するだけです．

taro.age = 7 ;

この「構造体.メンバー名」という形式は，taro のような単独の変数だけではなく，構造体の配列要素に対しても同様に使用できます．たとえば，

family [0] .sex

と書くと，「family [0] の sex」が指定されるわけです．

●ポインタを使った構造体の操作

すでに述べた int 型や char 型の変数と同様に，構造体もポインタを使って扱うことができます．とくに構造体を扱う関数を作る場合には，構造体へのポインタは不可欠です．なぜなら，MSX-C では構造体のような複合データは，ポインタの形にしなくてはパラメータとして受け渡すことができないからです．

その例として，struct person 型の構造体変数の内容を表示する whatis() という関数の定義を図 10.16 に示しましたのでご覧ください．

```
whatis(who)
struct person    *who; ············ whoはstruct person型へのポインタ
{
    printf("Name: %s¥n", (*who).name);
    printf("Sex:  %c¥n", (*who).sex);
    printf("Age:  %d¥n", (*who).age);
}
```

図 10.16 構造体へのポインタをパラメータとする関数

この関数を使って，たとえば taro のデータを表示するには，

whatis(&taro) ;

という関数呼び出しを行えばよいのです．すると whatis()のパラメータ who には，構造体 taro へのポインタが渡されますから，あとは間接演算子「*」を使ってそのポインタの指し示す内容を調べ，画面に表示するだけです．

ところで図 10.16 では，who の指し示す構造体のメンバーを，ここまでの説明に従って，

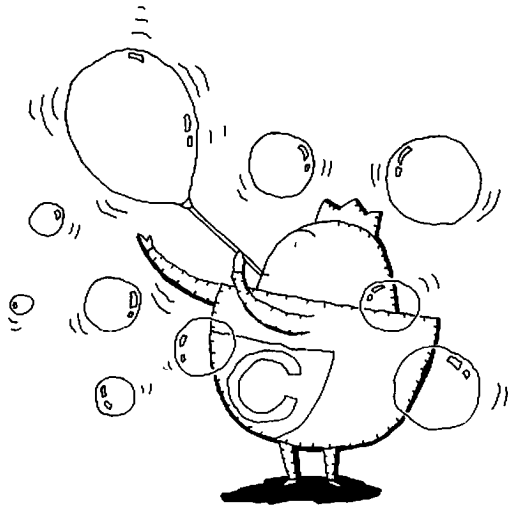
```
(* who).name
```

という形式で指定しましたが，このような「ポインタの指し示す構造体のメンバー」を表すには，もう 1 つ次の書式が用意されています．

```
who->name
```

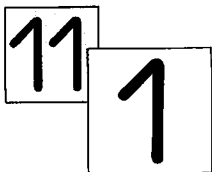
この「(* who).name」と「who->name」は，まったく同じ意味ですが，実際には後者のほうがよく用いられます．

11章

データの保存と利用

最近，ペーパーレス(Paper-less)環境，という言葉をよく聞くようになりました．要するに，すべての文書がコンピュータのファイルに収められ，それまで会社の中を埋め尽くしていた書類の山がきれいさっぱりなくなってしまうという，理想のオフィスを表す言葉であります．

しかし，机の上から紙の山が消えることのどこが「理想」なのか？東南アジアや南米アマゾン流域の緑の資源を食い潰さなくなり，宇宙船地球号の未来が少しでも明るくなるから？　いやいや，そんなきれいごとでは企業の論理に対抗はできません．ペーパーレス環境には，もっと現実的なメリットがあるのです．

データをコンピュータファイルに収めるということは，それをコンピュータが直接処理可能な形態にするということです．紙の上に印刷されたデータは，単にそれを眺めることしかできませんが，コンピュータファイル化されたデータは，それをもとにグラフも描ければ，将来の予測をすることもできます．つまり，データを何倍にも活かして使うことが可能となるのです．

C言語は，このようなファイルの使い回しが非常に得意なプログラミング言語です．そのささやかな第一歩として，この章では文字列データを使ったファイル操作の基本について紹介することにしましょう．

11-1 ファイルを扱うプログラム

ここではファイルへのデータ書き込みと，ファイルからのデータ読み出しの方法について説明します．この２つの処理は，基本的に次のような流れで行われます．

① ファイルのオープン
② データの読み書き
③ ファイルのクローズ

これはBASICでファイルのデータを操作する場合とまったく同じです．

データ読み書きの基本型

ここではファイル入出力の基本的な方法を説明します．

●データ書き込みの基本型

ファイルへの書き込みを行うには，まず目的のファイルを作成し，書き込みの準備を行わなくてはなりません．この操作を「ファイルのオープン」といいます．

ファイルのオープンには，fopen()という関数を使用します．たとえば"test"という名前のファイルをオープンするには，次のような形式でfopen()を呼び出します．

```
fp = fopen("test", "w") ;
```

このとき変数fpに代入されるfopen()の戻り値は，個々のファイルを区別する認識番号のようなもので，正式には「ファイル構造体へのポインタ」と呼ばれる特殊な型のデータですが，本書では単にファイルポインタと呼ぶこ

とにします．ファイルポインタは，次のような形式で宣言しておく必要があります．

FILE　＊fp；

以後のファイル操作は，ファイル名ではなく，このファイルポインタを使って行うことになります．たとえばファイルに文字列を書き込むには，

fputs(文字列, fp)；

という関数呼び出しを行います．すでにおなじみのputs()は画面に文字列を表示するものでしたが，このfputs()という関数は，同様な処理をfpで指定されたファイルに対して行うものと考えればよいでしょう．

必要な回数だけfputs()を実行してデータを書き込み終えたら，ファイルをクローズしなければなりません．ファイルのクローズは，fclose()という関数を使って，次のように行います．

fclose(fp)；

以上の作業を実際のプログラムにまとめたものを，図11.1に示します．

```
#include <stdio.h>
main()
{
    FILE  *fp; ……………ファイルポインタの宣言

    fp = fopen("test", "w"); ……………ファイルのオープン
    fputs("data-1¥n", fp);   ……………データの書き込み
    fputs("data-2¥n", fp);   ……………データの書き込み
    fclose(fp); ……………ファイルのクローズ
}
```

図11.1　ファイルへの文字列書き込みプログラム〝WTEST.C〟

図11.1のプログラムを実行すると，〝TEST〟というファイルが作成されますから，その内容をTYPEコマンドで表示して確かめてください．次のように2行ぶんのデータが書き込まれているはずです．

```
A>type test
data-1
data-2
```

図 11.2 図 11.1 の実行結果を確認する

●データ読み出しの基本型

ファイルからのデータ読み出しも，上のデータ書き込みの例と同様にファイルポインタを利用して行います．

まず，データ読み出しのためにファイルをオープンするには，fopen()を次のような形で実行します．

```
fp = fopen("test", "r") ;
```

さきほどは，fopen()の第2パラメータには，書き込み(write)モードを示す"w"を指定しましたが，今度は読み出し(read)モードを示すために，"r"というパラメータを指定するわけです．

こうしてオープンされたファイルから文字列の読み込みを行うには，fgets()を使用します．これはキーボードからの文字列入力関数 gets()のファイル版と考えてください．

ファイルポインタ fp で示されるファイルから文字列を読み込み，char 型の配列に代入するには，

```
fgets(配列名, 文字数, fp) ;
```

という関数呼び出しを行います．このとき配列には改行コードまでの1行ぶんのデータが読み込まれます．読み込んだデータの最後には自動的にヌル文字'¥0'が付け加えられますから，これを通常の文字列として扱うことが可能です．なお，読み出そうとした行が指定した文字数にはいりきらない場合は，残りのデータは次回の fgets()で読み込まれます．

データの読み出しが終わったなら，最後にやはり，

fclose(fp);

を実行して，ファイルをクローズします．

以上の操作を実際のプログラムにまとめたものを図 11.3 に示します．これは図 11.1 で作った〝test〟ファイルの内容を読み出して画面に表示するものです．

```
#include <stdio.h>
main()
{
    char  oneline[81];
    FILE  *fp; ……………ファイルポインタの宣言

    fp = fopen("test", "r"); ……………ファイルのオープン
    fgets(oneline, 81, fp);  ……………1行目のデータの読み出し
    puts(oneline);           ……………画面に表示
    fgets(oneline, 81, fp);  ……………2行目のデータの読み出し
    puts(oneline); ……………画面に表示
    fclose(fp);    ……………ファイルのクローズ
}
```

図 11.3　1 行ずつデータを読み出すプログラム

●データ読み出しのチェックポイント

いま見ていただいた図 11.3 は，2 行のデータを含む〝test〟というファイルを読み出すためだけのプログラムでしたが，これを元にして，指定した任意のファイルの内容を表示するプログラムを作ってみます．

このとき，次の 2 つの問題を考える必要があるでしょう．

① 存在しないファイル名が指定されたらどうするか
② ファイルの最後をどう認識するか

1 つ目の問題は，fopen()の戻り値を調べることで解決できます．指定したファイルがオープンできなかった場合，fopen()は NULL という記号で表される特別な値(NULL ポインタ)を返しますから，次のようなプログラムを書けばよいのです．

```
fp = fopen(ファイル名, "r");
if (fp == NULL) {
    ファイルがオープンできなかった場合の処理;
}
```

2つ目の問題は，fgets()の戻り値で判断できます．ファイルにもう読み出すデータがなくなった状態でfgets()を実行すると，fgets()はやはりNULLを返します．ですから次のようにfgets()の戻り値がNULLかどうかをチェックしていれば，ちょうどファイルのデータをすべて読み終えたところでループを終了させることができます．

```
while (fgets(…, …, …) != NULL) {
    読み込んだデータの処理;
}
```

図 11.4 は，上で述べた方法を使って作成した，任意のファイルの内容を表示するプログラム "newtype.c" です．

```
#include <stdio.h>
main(argc, argv)
int  argc;
char *argv[ ];
{
    char  oneline[81];
    FILE  *fp;

    fp = fopen(argv[1], "w");
    if (fp == NULL) {
        printf("¥nﾌｧｲﾙ '%s' ｶﾞ ｵｰﾌﾟﾝ ﾃﾞｷﾏｾﾝ¥n", argv[1]);
        return;
    }

    while (fgets(oneline, 81, fp) != NULL)
        puts(oneline);

    fclose(fp);
}
```

図 11.4 指定ファイルの内容を画面に表示するプログラム

このプログラムを使用するには，コマンドラインから，

A>newtype ファイル名

と指定します．

●データ書き込みのチェックポイント

ファイルへのデータ書き込みを行うときも，上で述べたことと同様なエラーチェックの問題はおこります．この場合の問題点は次の2つです．

① 不正なファイル名("S.O.S" など)が指定されたらどうするか
② ディスクにデータが書き込めなかったらどうするか

1つ目は，やはりfopen()の戻り値で判断できます．書き込みモードでファイルをオープンしたとき，もし正常にオープンできなければfopen()はNULLを返してきますから，読み出しの場合と同様に次のようなプログラムで対処可能です．

```
fp = fopen(…, …);
if (fp == NULL) {
    ファイルがオープンできなかった場合の処理;
}
```

2つ目のデータが書き込めないという問題は，まずほとんど，ディスクが一杯になった場合です．これは通常はfputs()を実行した時点でおこるのですが，ごくまれに最後のデータを書き込んだ後fclose()を実行した時点でディスクが満杯になったことがわかる場合もありますから，その両方でエラーチェックを行わなければなりません．

fputs()は，データが正常に書き込めないと，EOFという記号で表される特別な値を返します．また，fclose()も同様に，最後のデータが書き込めないときにはEOFを返します．そこで，次のようなプログラムを作ることになります．

```
if (fputs(…, …) == EOF) {
    ディスクが一杯になった場合の処理;
}

if (fclose(…) == EOF) {
    ディスクが一杯になった場合の処理;
}
```

図 11.5 に，これらのエラーチェックをマジメに行うプログラムの例を示します．このプログラムは，指定したファイルに "test-data¥n" という文字列を 6 万回書き込むものですが，すべて書き込み終える前に，たぶん「データが書き込めません」というエラーメッセージを表示して中断するでしょう．これは fputs()の実行時にディスクが満杯になったのです．

あるいは，もしかしたら「ファイルがクローズできません」というメッセージが表示されるかもしれません．それは最後のデータを書き込んだ後の fclose()でディスクが満杯になったことが判明した場合です．

```
#include <stdio.h>
main(argc, argv)
int  argc;
char *argv[ ];
{
    unsigned  i;
    FILE      *fp;

    fp = fopen(argv[1], "w");
    if (fp == NULL) {
        printf("¥nﾌｧｲﾙ '%s' ｶﾞ ｵｰﾌﾟﾝ ﾃﾞｷﾏｾﾝ¥n", argv[1]);
        return;
    }

    for (i = 1; i <= 60000; ++i) {
        if (fputs("test-data¥n", fp) == EOF) {
            puts("¥nﾃﾞｰﾀ ｶﾞ ｶｷｺﾒﾏｾﾝ¥n");
            return;
        }
        printf("%8u", i);
    }
```

```
    if (fclose(fp) == EOF) {
        printf("¥nﾌｧｲﾙ '%s' ｶﾞ ｸﾛｰｽﾞ ﾃﾞｷﾏｾﾝ¥n", argv[1]);
        return;
    }

    puts("¥nOK¥n");
}
```

図 11.5 書き込み時のエラーチェックの例

読み書きの手段のバリエーション

ここまで述べたとおり，ファイルに対する文字列の読み書きは，fgets()とfputs()で行うことができました．これ以外にもMSX-Cには，1文字単位でファイル入出力を行うgetc()とputc()，そして数値の入出力を行うfprintf()とfscanf()などが用意されています．

これらの関数を使う場合も，ファイルのオープン／クローズはfgets()，fputs()と同様に行います．それぞれのデータの入出力の手順は，以下のとおりです．

●ファイルへの1文字出力 —— putc()

putc()は，ファイルポインタで示されるファイルに1文字ぶんのchar型データを書き込みます．正常に書き込めなかった場合は，EOFを返します．putc()の一般的な使い方は次のようになります．

```
if (putc(c, fp) == EOF) {
    データが書き込めなかった場合の処理;
}
```

●ファイルからの1文字入力 —— getc()

getc()は，ファイルポインタで示されるファイルから1文字ぶんのデータを読み出し，その文字コードを戻り値として返します．また，ファイルの最

後までデータが読み出されている場合，さらに読み出しを行うと，getc()はEOFを返します．getc()を使ってファイルの最後まで読み出すループを作ると次のようになります．

```
int  c;
while ((c = getc(fp)) != EOF)
    putchar((char)c);
```

ここで，getc()の戻り値を受け取る変数がint型で宣言されていることに注意してください．これは，EOFの値がchar型の表現できる範囲に入っていないためです．

●ファイルへの数値出力 ―― fprintf()

fprintf()は，画面に対してprintf()が行うのとまったく同様な動作を，ファイルに対して行います．データが正常に書き込めなかった場合はEOFを返します．

fprintf()によって変数iの値をファイルfpに出力するには，次のような関数呼び出しを行います．

```
fprintf(fp, "%d", i);
```

要するに，形式としてはprintf()の1番目のパラメータにファイルポインタが追加されただけです．これ以外にも，printf()が画面に対して表示できることはすべて，fprintf()を使ってファイルに出力することが可能です．

●ファイルからの数値入力 ―― fscanf()

fprintf()がprintf()の出力を画面からファイルに振り向けたものならば，fscanf()はscanf()という関数の入力をキーボードからファイルに振り向けたものに相当します．データがない場合には，fscanf()はEOFを返します．

この関数でファイルfpから変数iに整数データを読み込むには，

```
fscanf(fp, "%d", &i);
```

という呼び出しを行います．

特別なファイルの利用法

前の項では，ディスク上のファイルに対する入出力の方法を紹介しましたが，プリンタ出力や画面出力，キーボード入力などのいわゆるデバイスについても，まったく同様の方法でデータ入出力のプログラムを書くことができます．

●プリンタへの出力 —— "prn" という名の特殊ファイル

これはMSX-Cというよりも，MSX-DOSがもともと持っている機能なのですが，"prn"というファイルに書き込みを行うと，ディスク上の"prn"というファイルではなく，プリンタに対してデータが出力されます．このことを利用すると，プリンタへの打ち出しのプログラムが作成できます．

図11.6に，プリンタにAからZまでの26文字を打ち出すプログラム例を示します．まず，

```
fp = fopen("prn", "w") ;
```

という呼び出しでプリンタをオープンしたあと，ここではputc()を使っていますが，fputs()やfprintf()などでデータを出力してもかまいません．

```
#include <stdio.h>
main()
{
    FILE  *fp;
    char  c;

    fp = fopen("prn", "w"); ……………プリンタをオープンする
    for (c = 'A'; c <= 'Z'; ++c)
        putc(c, fp);
}
```

図11.6 プリンタのオープン

● コンソール入出力 —— "con" という名の特殊ファイル

プリンタからデータを読み出すことはできませんから，上記の "prn" デバイスファイルは書き込みモードでしかオープンする意味はありません．

一方 "con" という名前で表される特殊ファイルは，読み出し／書き込みの両方向でオープンすることが可能です．これは，読み出しモードの場合はキーボード入力，書き込みモードの場合は画面出力が指定されます．

たとえば，"con" ファイルをオープンして，

```
fp = fopen("con", "w") ;
fputs("test¥n", fp) ;
```

という書き込みを行うと，これはちょうど，

```
puts("test¥n") ;
```

を実行した場合と同様に，画面に対して文字列 "test" が出力されることになります．

ただし，"con"に対する fputs() と通常の puts() は，まったく同じものというわけではありません．それは，puts()のほうは単なる画面ではなく，「標準出力」にデータを出力するものだからです．

● stdin と stdout

Cには，プログラムを作る上でたいへんに便利な「標準入力」，「標準出力」と呼ばれる概念が存在します．

この標準入出力は，通常はコンソール入出力，すなわちキーボード入力と画面出力なのですが，プログラムの実行時にリダイレクションを行うと，そのリダイレクション先に指定されたファイルを対象としてデータを操作するようになります．

この2つには，プログラムを起動した時点で次のような名前のファイルポインタが割り当てられます．

stdin …… 標準入力
stdout …… 標準出力

これらに対するファイルのオープンは自動的に行われるため，プログラム中で実行する必要はありません．たとえば，標準出力に "data" というデータを出力するプログラムは，図 11.7 のようになります．

```
#include <stdio.h>
main()
{
        fputs("data", stdout);
}
```

図 11.7 標準出力へのデータ書き込みプログラム

実行してみるとわかるように，このプログラムは通常は画面にデータを表示するのですが，出力リダイレクションを行うと，リダイレクション先のファイルにデータを書き込みます．

これは，実際のところ，

puts("data") ;

の動作とまったく同じものです．つまり，puts() とは「標準出力に対する fputs()」の省略型にほかならないわけです．

これ以外にも MSX-C では，**表 11.1** に示した左右の関数は，双方まったく同じ動作を行います．

入出力先を明示的に指定	入出力先を省略
fputs(….stdout);	puts(…);
putc(….stdout);	putchar(…);
fprintf(stdout,…,…);	printf(…,…);
fgets(…,…,stdin);	gets(…,…);
getc(stdin);	getchar();
fscanf(stdin,…,…);	scanf(…, …);

表 11.1 標準入出力を使った動作の同じ関数の対応表

● stderr の利用法

標準出力を使うと，リダイレクションの状態によって，データの出力先を画面にするかファイルにするか，自動的に振り分けてくれるのは非常に便利なのですが，1つだけ困る場合があります．

それは，エラーメッセージの表示です．エラーメッセージというものは，そもそも目に見えるように表示されてこそ意味があるわけですが，puts()やprintf()を使ったプログラムでは，出力リダイレクションを行ったとき，エラーメッセージまでファイルに書き込まれ画面に現れてくれません．

この問題を避けるには，標準出力とは別に，もう1つ〝con〟ファイルを書き込みモードでオープンして，画面に表示したいメッセージはつねにそちらへ出力すればよいはずです．

実はそのために，Cにはあらかじめ標準エラー出力というものが用意されています．これはstderrというファイルポインタで表され，stdinやstdoutと同様に，いちいちファイルをオープンしなくとも利用することが可能です．

図11.8は，この標準エラー出力の使用例です．このプログラムをコンパイルし，適当なファイルに出力をリダイレクトしてみてください．puts()で出力するデータは指定したファイルに書き込まれますが，「データが書き込めません」というエラーメッセージは，出力リダイレクションを行っているにも関わらず，画面に表示されるはずです．

```
#include <stdio.h>
main()
{
    unsigned  i;

    for (i = 1; i <= 6ØØØØ; ++i) {
        if (puts("test-data¥n") == EOF) {
            fputs("¥nﾃﾞｰﾀ ｶﾞ ｶｷｺﾒﾏｾﾝ¥n", stderr);
            return;
        }
    }
    puts("OK¥n");
}
```

図11.8　標準エラー出力の利用

Appendix 1
MSX-C エラーメッセージ

● MSX-C　CF コマンドエラーメッセージ

- **array of function**　関数の配列は許されない
- **bad #???**　#コマンドが間違っている(#コマンドが式中にあるか，プリプロセッサでないコマンドを指定した)
- **bad #〈プリプロセッサ命令〉 statement**　#〈プリプロセッサ命令〉が間違っている
- **bad abstract declarator**　抽象宣言子が正しくない
- **bad assignment**　代入が間違っている
- **bad cast**　キャスト演算が間違っている(配列，構造体，共用体に対してはキャストできない)
- **bad character**　ソースプログラム中に不適当な文字(コントロール文字など)がある
- **bad condition**　条件式が不正である
- **bad conditional expression (missing '?')**　条件式で3項演算子に'?'がない
- **bad constant expression**　定数式が必要なところに変数や関数が使われている
- **bad drive :**　ドライブの指定が間違っている
- **bad element**　文が存在しない，あるいは不正である(ラベルや while の後ろなど)
- **bad element encountered externally**　関数の外側に不正な文がある
- **bad indirection**　間接指定が間違っている(ポインタに*を多く付けすぎた)
- **bad number**　数値表現に範囲外の文字が使われている(8進数に0～7以外の文字を使った場合など)
- **bad option: 〈文字〉**　オプションの指定が間違っている('－〈文字〉'というオプションは存在しない)
- **bad parameter list**　引数リストが間違っている(引数付きマクロの定義中，仮引き数が不正か，仮引数の区切りが','でない)
- **bad parameter type**　関数呼出しの際の引数の型が数値型ではない(配列，構造体，共用体を渡すにはポインタを使う)
- **bad parameter type '〈仮引数〉'**　関数定義の際の〈仮引数〉の型が数値型ではない
- **bad storage class**　指定の記憶クラスはここでは使用できない
- **bad switch expression**　switch 文に誤りがある(switch の式には数値型の結果が必要)
- **bad table ratio**　'－r'オプションで指定された比率が正しくない
- **bad type in cast**　キャストする型が間違っている

- **'break' outside switch/break**　breakがswitch文の外にある
- **cannot initialized**　自動(auto)変数の配列を初期化をしようとしている
- **cannot make ［<file>］**　<file>が作成できない
- **cannot open ［<file>］**　ファイル<file>がオープンできない
- **'case' after default**　defaultより後ろにcaseがある(MSX-Cではdefaultより後ろにcaseを書くことはできない)
- **'case' outside switch**　switchの外側にcaseがある
- **conflict definition '<識別子>'**　<識別子>の宣言が一致しない(2つ以上の外部・前方参照の宣言で，関数や変数の宣言が一致していない)
- **conflicting number of macro parameter**　マクロの引数の数が合わない
- **'continue' outside loop**　while、forなどのループの外にcontinueがある
- **'default' appeard twice**　1つのswitch文でdefaultが2度使われている
- **'default' outside switch**　defaultがswitch文の外にある
- **disk full**　ディスクがいっぱいである
- **'double' not supported**　double型は扱えない
- **duplicate 'case' label**　switchでcaseの定数値の同じものが2つ以上ある
- **duplicate label '<ラベル>'**　同一関数内で<ラベル>が2か所以上存在する
- **duplicate member**　同一の構造体，共用体の中に同じメンバ名が存在する
- **duplicate storage class**　記憶クラスが2つ以上指定されている
- **duplicate tag '<タグ>'**　同じ<タグ>で構造体や共用体が定義されている
- **duplicate type specifier**　変数の型が2つ以上指定されている
- **'(' expected**　'('が必要
- **')' expected**　')'が必要
- **':' expected**　'：'が必要
- **';' expected**　'；'が必要
- **'］' expected**　'］'が必要
- **'float' not supported**　float型は扱えない
- **function cannot appear**　関数は指定できない
- **function expected**　未定義の関数を呼び出している
- **function returns structured data**　関数は構造体を返すことができない
- **hash table over flow**　ハッシュテーブルがオーバーフローした
- **heap over flow**　ヒープがオーバーフローした
- **improper function mode**　ファンクションモードが不適当である(nonrec、recursiveが変数に対して使われている)
- **# include too nested**　#includeのネストは4重まで
- **known dimension expected**　配列の大きさは定数で指定しなければならない
- **'long' not supported**　long型は扱えない

・**l-value required** 左辺値が必要(記憶位置を示す式を左辺値と呼ぶ．値を代入するには変数の名前など，記憶位置を示す式が必要である)
・**missing '}'** '}'が必要
・**missing condition** 条件が必要
・**missing identifier** 識別子(変数名、関数名、配列名など)が必要
・**missing member name** 構造体変数や構造体変数へのポインタに続くメンバ名がない
・**missing operator (or semicolon)** 演算子か';'が必要
・**missing quote** シングルクォートまたはダブルクォートが必要
・**missing tag** タグ名が必要
・**not appear in parameter list '〈仮引数〉'** 〈仮引数〉は引数リストにない
・**pointer type mismatch - use cast** ポインタの型が合っていないのでキャスト演算子を使え
・**pool over flow** プールがオーバーフローした('-r'オプションでワークテーブルの比率を修正せよ)
・**precedence error** 優先順位が正しくない
・**redeclaration of '〈識別子〉'** 〈識別子〉が再定義されている
・**sizeof(func) not permitted** sizeof(func)は許可されていない
・**sorry, too small memory** メモリ不足によりコンパイル不可能
・**stac over flow** stacが足りない('-r'オプションでワークテーブル比率を修正せよ)
・**static function '〈関数〉' not defined** スタティックな関数〈関数〉は定義されていない
・**symbol table over flow** シンボルテーブルが足りない('-r'オプションでシンボルテーブルの比率が大きくなるよう修正せよ)
・**syntax error** この文または前の文が正しくない
・**too many initializers** 初期化要素の数が多すぎる
・**too much parameters** パラメータが多すぎる(引数付きマクロの引数は11まで)
・**type mismatch** 型が合っていない
・**undeclared function '〈関数〉'** 〈関数〉は宣言されていない
・**undeclared identifier '〈識別子〉'** 未定義の〈識別子〉を使用している
・**undeclared member name '〈メンバ〉'** 〈メンバ〉が宣言されていない
・**undeclared parameter '〈仮引数〉'** 仮引数が宣言されていない
・**undefined label '〈ラベル〉' in function '〈関数〉'** 〈関数〉には〈ラベル〉が定義されていない
・**undefined struct/union** 未定義のstructまたはunion型として宣言されている
・**unexpected eof in this comment** コメントの途中でファイルが終わっている
・**unexpected eof inside function '〈関数〉'** 〈関数〉が完結する前にソースファイルが終っている
・**unexpected eof** ディスク上にテンポラリファイルのための十分な領域がない

- **useless expression** 演算結果が使われていない
- **'while' expected** while がない(do while の while がループのブロックが終わっても見つからない)

● CG コマンドエラーメッセージ

- **bad option: 〈文字〉** 存在しないスイッチ'ー〈文字〉'が指定されている
- **cannot make: 〈ファイル〉** 〈ファイル〉が作成できない
- **cannot open: 〈ファイル〉** 〈ファイル〉が見つからない
- **fuction overflow in "〈関数〉" ...specify less than 〈数〉 by -r** 〈関数〉でコード生成用の領域がなくなった，'-r'オプションで〈数〉より小さい数を指定し直せ
- **illegal .tco file at "〈関数〉" 〈行〉:〈桁〉** .tco ファイルが壊れてる(再度，CF から始める)
- **sorry, too small memory** メモリ不足によりコンパイル不可能
- **stack overflow** スタックが足りない
- **symbol table overflow in "〈関数〉" ...specify more than 〈数〉 by -r** 〈関数〉でシンボルテーブルがなくなった，'-r'オプションで〈数〉より大きい数を指定し直せ
- **warning: >> more than 8 bit in "〈関数〉"** 警告：8 ビット以上右シフトしている
- **warning: << more than 8 bit in "〈関数〉"** 警告：8 ビット以上左シフトしている
- **warning: << more than 16 bit in "〈関数〉"** 警告：16 ビット以上左シフトしている
- **warning: too big char constant 〈数〉 in "〈関数〉"** 警告：キャラクタ定数の値が大きすぎる

Appendix 2
VRAMディスクの使い方

以下に第2章で述べたVRAMディスクのプログラムを紹介します．なお，このプログラムはMSX-DOS1の上でしか利用できません．

● "RAMDSK.COM"の作成方法

プログラムは16進ダンプリストの形式で掲載しました(図A.1)．これをCOMファイルに変換するには，リストA.2に示す"MAKECOM.C"をコンパイルして使用します．最終的にVRAMディスクプログラム"RAMDSK.COM"を手に入れるまでの手順は以下のとおりです．

```
A>med makecom.c ……………"MAKECOM.C"を入力する
       :
A>cc makecom    ……………コンパイルして"MAKECOM.COM"を作る
       :
A>med ramdsk.x  ……………"RAMDISK.X"を入力する
       :
A>makecom ramdsk.x ramdsk.com ……………"RAMDISK.COM"を作る
```

図A.1　RAMDISK.COMを手に入れる手順

● VRAMディスクの利用法

MSX-DOSのコマンドラインから，

```
A>ramdsk -f
```

と入力すると，VRAMディスクがCドライブに割り当てられ，初期化されます．通常はアドレス08000hから1FFFFhまでの96KバイトのVRAMが使

用されますが，次の形式で，VRAMに割り当てる先頭のアドレスと使用するメモリのサイズを指定することもできます．

A>ramdsk -f -t90 -s170 ←09000h以降の17000hバイトを使用
(100hバイト単位の16進数で指定する)

● VRAMディスクの環境設定

このVRAMディスクは，起動するときにMSX-DOSの動作環境をリセットするため，"AUTOEXEC.BAT"の中で使うと，それ以降のバッチコマンドが無視されてしまいます．リストA.3に示すAUTOKEYコマンドは，それを避けるために使用するものです．これもやはりエディタで入力してから，MAKECOMコマンドで次のようにCOMファイルに変換してください．

A>makecom autokey.x autokey.com

このAUTOKEYコマンドは，DOSのコマンドをセミコロンで区切ってパラメータとして与えると，それらを自動的に順次実行します．たとえば，

A>autokey mode 40;dir;

と入力すると，まず「mode 40」，続いて「dir」が実行されます．

この機能を使い，下図のような"AUTOEXEC.BAT"および"INITDSK.BAT"を作るとよいでしょう．"AUTOEXEC.BAT"がVRAMディスクの起動を行い，それ以降のファイルコピー作業は"INITDSK.BAT"が行うわけです．

```
autokey ramdsk -f;initdsk; ··············VRAMディスクを起動してINITDSKを実行
```

図A.2 AUTOEXEC.BAT

```
copy a:command.com c: ··············これは絶対に必要
copy a:med.com c:     ··············以下VRAMディスクに置きたいファイルを
c:                                  適当にコピーする
```

図A.3 INITDSK.BAT

```
Ø1ØØ  C3 B7 Ø4 3A FC FA E6 Ø6 ØF 47 Ø4 3A 63 Ø7 4F 3A  721
Ø11Ø  E3 Ø8 81 28 Ø2 38 11 1Ø Ø5 FE 21 D8 18 ØA 1Ø Ø5  532
Ø12Ø  FE 81 D8 18 Ø3 B7 C8 DØ 11 66 Ø2 C3 81 Ø1 21 8Ø  84Ø
Ø13Ø  ØØ 46 23 Ø4 CD FD Ø1 D8 CD Ø8 Ø2 28 F7 FE 3F 28  79B
Ø14Ø  3D FE 2F 2Ø Ø2 18 Ø4 FE 2D 2Ø 33 CD 5Ø Ø1 18 E4  68Ø
Ø15Ø  CD FD Ø1 38 29 CD 17 Ø2 FE 46 28 16 FE 52 28 18  774
Ø16Ø  FE 54 28 2Ø FE 53 28 44 11 79 Ø2 ØE Ø9 CD Ø5 ØØ  62C
Ø17Ø  18 ØC 3E FF 32 61 Ø7 C9 3E FF 32 62 Ø7 C9 11 E9  7CF

Ø18Ø  Ø2 C3 D6 Ø4 CD FD Ø1 38 F5 CD 2Ø Ø2 38 FØ CD D9  9D4
Ø19Ø  Ø1 7A FE Ø2 3Ø ØC CB 43 2Ø ØD 1F CB 1B 7B 32 E3  717
Ø1AØ  Ø8 C9 11 8A Ø2 18 DA 11 A2 Ø2 18 D5 CD FD Ø1 38  7A5
Ø1BØ  CD CD 2Ø Ø2 38 C8 CD D9 Ø1 7A FE Ø2 3Ø 16 B7 2Ø  8AA
Ø1CØ  ØA 7B FE ØC 3Ø Ø5 11 D3 Ø2 18 B6 CB 3A CB 1B 7B  79E
Ø1DØ  32 63 Ø7 C9 11 BF Ø2 18 A8 CD F5 Ø1 16 ØØ 5F CD  7CC
Ø1EØ  FD Ø1 38 21 CD 2Ø Ø2 D8 CD F5 Ø1 EB 29 29 29 29  85Ø
Ø1FØ  EB B3 5F 18 EA D6 3Ø FE ØA D8 D6 Ø7 C9 1Ø Ø2 37  9C4

Ø2ØØ  C9 7E 23 B7 C9 Ø4 2B C9 FE Ø9 C8 FE ØD C8 FE ØA  A8C
Ø21Ø  C8 FE 2Ø C8 FE 3B C9 FE 61 D8 FE 7B DØ D6 2Ø C9  CFF
Ø22Ø  CD 17 Ø2 FE 3Ø D8 FE 3A 38 Ø9 FE 41 D8 FE 47 38  A19
Ø23Ø  Ø2 37 C9 B7 C9 2E ØØ 67 16 64 CD 45 Ø2 16 ØA CD  7C2
Ø24Ø  45 Ø2 7C 18 15 1E ØØ 7C BA 38 Ø4 1C 92 18 F9 67  6E6
Ø25Ø  7B B7 2Ø Ø6 7D B7 C8 28 Ø4 AF 2C C6 3Ø 5F E5 ØE  8F3
Ø26Ø  Ø2 CD Ø5 ØØ E1 C9 42 61 64 2Ø 41 6C 6C 6F 63 61  851
Ø27Ø  74 69 6F 6E 2Ø 21 ØD ØA 24 42 61 64 2Ø 4F 7Ø 74  7ØØ

Ø28Ø  69 6F 6E 2Ø 21 ØD ØA ØD ØA 24 54 6F 7Ø 2Ø 61 64  671
Ø29Ø  64 2Ø 69 73 2Ø 74 6F 6F 2Ø 68 69 67 68 2Ø 21 ØD  77Ø
Ø2AØ  ØA 24 54 6F 7Ø 2Ø 61 64 64 2Ø 62 69 74 2Ø 23 38  724
Ø2BØ  2Ø 6D 75 73 74 2Ø 62 65 2Ø 3Ø 2Ø 21 ØD ØA 24 53  69F
Ø2CØ  69 7A 65 2Ø 69 73 2Ø 74 6F 6F 2Ø 62 69 67 2Ø 21  8Ø9
Ø2DØ  ØD ØA 24 53 69 7A 65 2Ø 69 73 2Ø 74 6F 6F 2Ø 73  7A7
Ø2EØ  6D 61 6C 6C 2Ø 21 ØD ØA 24 2D 2Ø 53 65 74 2Ø 56  6F1
Ø2FØ  52 41 4D 2Ø 64 69 73 6B 2Ø 64 72 69 76 65 72 2Ø  867

Ø3ØØ  2D ØD ØA 2Ø 2Ø 55 73 61 67 65 3A 2Ø 72 61 6D 64  777
Ø31Ø  73 6B 2Ø 5B 6F 7Ø 74 69 6F 6E 73 5D ØD ØA 2Ø 2Ø  829
Ø32Ø  5B 6F 7Ø 74 69 6F 6E 73 5D ØD ØA 2Ø 2Ø 2Ø 2Ø 2D  7A8
Ø33Ø  72 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 3A 2Ø 52 65 6D 6F 76 65 2Ø  74A
Ø34Ø  56 52 41 4D 2Ø 64 69 73 6B ØD ØA 2Ø 2Ø 2Ø 2Ø 2D  7Ø5
Ø35Ø  66 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 3A 2Ø 46 6F 72 6D 61 74 2Ø  759
Ø36Ø  56 52 41 4D 2Ø 64 69 73 6B ØD ØA 2Ø 2Ø 2Ø 2Ø 2D  725
Ø37Ø  74 3C 68 65 78 3E 2Ø 3A 2Ø 54 6F 7Ø 2Ø 61 64 64  899

Ø38Ø  2Ø 28 7Ø 61 72 2Ø 31 3Ø 3Ø 68 29 ØD ØA 2Ø 2Ø 2Ø  6C4
Ø39Ø  2Ø 2D 73 3C 68 65 78 3E 2Ø 3A 2Ø 53 69 7A 65 2Ø  844
Ø3AØ  28 7Ø 61 72 2Ø 31 3Ø 3Ø 68 29 2Ø ØD ØA 24 56 52  75Ø
Ø3BØ  41 4D 2Ø 64 69 73 6B 2Ø 64 72 69 76 65 72 2Ø 61  936
Ø3CØ  6C 72 65 61 64 79 2Ø 65 78 69 73 74 2E 2E 2E ØD  925
Ø3DØ  ØA 24 54 68 65 72 65 2Ø 69 73 2Ø 6E 6F 74 68 69  934
Ø3EØ  6E 67 2Ø 74 6F 2Ø 72 65 6D 6F 76 65 2E 2E 2E ØD  8FD
Ø3FØ  ØA 24 44 72 69 76 65 2Ø 6E 61 6D 65 73 2Ø 61 72  93F
```

```
Ø4ØØ  65 2Ø 75 73 65 64 2Ø 75 7Ø 2E 2E 2E ØD ØA 24 44   844
Ø41Ø  65 66 61 75 6C 74 2Ø 64 72 69 76 65 2Ø 73 65 74   A37
Ø42Ø  74 6C 65 64 2Ø 6F 6E 2Ø 41 3A 2Ø 64 72 69 76 65   99B
Ø43Ø  2C ØD ØA 61 6E 64 2Ø 56 52 41 4D 2Ø 64 69 73 6B   8C7
Ø44Ø  2Ø 69 73 2Ø 2E 2E 2E 24 56 52 41 4D 2Ø 44 69 73   88Ø
Ø45Ø  6B 2Ø 44 72 69 76 65 72 2Ø 56 65 72 2Ø 31 2E 33   946
Ø46Ø  3Ø 2Ø 66 6F 72 2Ø 4D 53 58 32 ØD ØA 2Ø 2Ø 43 6F   84A
Ø47Ø  7Ø 79 72 69 67 68 74 2Ø 28 63 29 2Ø 31 39 38 37   944

Ø48Ø  2C 31 39 38 38 2Ø 62 79 2Ø 54 65 4B ØD ØA ØD ØA   7D3
Ø49Ø  2D 2Ø 73 65 74 74 6C 65 64 2Ø 6F 6E 2Ø ØØ 3A 2Ø   949
Ø4AØ  64 72 69 76 65 2Ø 28 2Ø 24 4B 2Ø 62 79 74 65 73   9D8
Ø4BØ  2Ø 29 2Ø 2D ØD ØA 24 CD 2E Ø1 CD Ø3 Ø1 3A 63 Ø7   7F2
Ø4CØ  CB 3F 3D 32 F2 Ø8 3A 67 F2 FE C3 3A 62 Ø7 2Ø ØC   B56
Ø4DØ  B7 2Ø 11 11 AE Ø3 ØE Ø9 CD Ø5 ØØ C7 B7 28 48 11   962
Ø4EØ  D2 Ø3 18 F2 31 ØØ C1 ØE ØD CD Ø5 ØØ F3 21 47 F3   AEC
Ø4FØ  35 2A 68 F2 E5 2E Ø2 11 67 F2 ED AØ ED AØ ED AØ   DCF

Ø5ØØ  11 7Ø F2 ED AØ ED AØ ED AØ 11 79 F2 ED AØ ED AØ   FBØ
Ø51Ø  ED AØ 11 ØF Ø4 ØE Ø9 CD Ø5 ØØ F3 E1 2E ØØ 7E 23   A4D
Ø52Ø  66 6F 22 4B F3 18 58 3A 47 F3 FE Ø7 38 Ø5 11 F2   B7E
Ø53Ø  Ø3 18 A3 31 ØØ C1 32 E4 Ø8 3C 32 47 F3 C6 4Ø 32   ADE
Ø54Ø  9D Ø4 11 48 Ø4 ØE Ø9 CD Ø5 ØØ 3A 63 Ø7 CB 3F CD   9A2
Ø55Ø  35 Ø2 11 A9 Ø4 ØE Ø9 CD Ø5 ØØ 3A C1 FC 21 Ø6 ØØ   94C
Ø56Ø  CD ØC ØØ 32 E1 Ø8 3A C1 FC 21 Ø7 ØØ CD ØC ØØ 3C   A88
Ø57Ø  32 E2 Ø8 CD 68 F3 CD 3Ø 4Ø 22 ØØ Ø8 CD 6B F3 EB   D31

Ø58Ø  21 FA ØA Ø1 ØØ Ø6 AF ED 42 EB ED 52 6F 3A 62 Ø7   BC6
Ø59Ø  B7 2Ø Ø7 24 24 22 4B F3 25 25 C5 D5 E5 7C 9Ø 4F   C3A
Ø5AØ  21 B7 Ø5 5E 23 56 23 7B B2 28 Ø5 1A 81 12 18 F3   A89
Ø5BØ  D1 C1 E1 D5 ED BØ C9 Ø2 Ø6 Ø8 Ø6 ØE Ø6 11 Ø6 21   BCØ
Ø5CØ  Ø6 3F Ø6 45 Ø6 4B Ø6 5F Ø6 66 Ø6 A5 Ø6 AB Ø6 B4   988
Ø5DØ  Ø6 C2 Ø6 C5 Ø6 C9 Ø6 D2 Ø6 D5 Ø6 1Ø Ø8 18 Ø8 1B   A3E
Ø5EØ  Ø8 45 Ø8 56 Ø8 5A Ø8 76 Ø8 93 Ø8 99 Ø8 A6 Ø8 AA   AØ7
Ø5FØ  Ø8 F8 Ø8 ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   6F8

Ø6ØØ  3A 62 Ø7 B7 28 Ø9 11 DC Ø6 CD C9 F1 C3 A9 Ø6 21   C98
Ø61Ø  E4 Ø8 7E 87 Ø6 ØØ 4F EB 21 55 F3 Ø9 73 23 72 11   BCC
Ø62Ø  Ø2 Ø8 21 67 F2 ED AØ ED AØ ED AØ 21 7Ø F2 ED AØ   F5B
Ø63Ø  ED AØ ED AØ 21 79 F2 ED AØ ED AØ ED AØ 21 ØB Ø8   FB1
Ø64Ø  22 68 F2 21 43 Ø8 22 71 F2 21 91 Ø8 22 7A F2 3E   C33
Ø65Ø  C3 32 67 F2 32 7Ø F2 32 79 F2 CD 68 F3 3A 61 Ø7   E99
Ø66Ø  B7 28 46 DD 21 E4 Ø8 21 ØØ ØØ DD 5E Ø3 DD 56 Ø4   CØ5
Ø67Ø  DD 46 ØC 19 1Ø FD 4D 44 21 ØØ Ø1 11 Ø1 Ø1 36 ØØ   9C1

Ø68Ø  ED BØ 21 ØØ Ø1 3E FE 77 23 3C 77 23 77 11 ØØ Ø1   B74
Ø69Ø  DD E5 CD 58 5Ø DD E1 11 ØØ ØØ DD 6E ØØ DD 66 ØC   E3Ø
Ø6AØ  CD 2Ø 47 11 1C Ø7 CD C9 F1 11 3C Ø7 CD 62 44 B7   DØD
Ø6BØ  28 ØB 11 E8 Ø6 CD C9 F1 CD 4E 54 18 EC 21 ØØ ØØ   CFD
Ø6CØ  22 5D Ø7 22 5F Ø7 23 22 4A Ø7 11 ØØ Ø1 CD 58 5Ø   9EB
Ø6DØ  11 3C Ø7 21 ØØ Ø6 CD B2 47 C3 ØØ Ø1 2Ø 72 65 6D   B39
Ø6EØ  6F 76 65 64 2E ØD ØA 24 ØD ØA 49 6E 73 65 72 74   B83
Ø6FØ  2Ø 44 4F 53 2Ø 64 69 73 6B 2Ø 6F 6E 2Ø 64 65 66   CØD
```

```
Ø7ØØ  61 75 6C 74 2Ø 64 72 69 76 65 2Ø 61 6E 64 2Ø 74  CD7
Ø71Ø  79 7Ø 65 2Ø 61 6E 79 2Ø 6B 65 79 24 2D 2Ø 56 52  C48
Ø72Ø  41 4D 2Ø 64 69 73 6B 2Ø 66 6F 72 6D 61 74 2Ø 63  CA5
Ø73Ø  6F 6D 7Ø 6C 65 74 65 2Ø 2D ØD ØA 24 ØØ 4D 53 58  BA6
Ø74Ø  44 4F 53 2Ø 2Ø 53 59 53 ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  965
Ø75Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  75Ø
Ø76Ø  ØØ ØØ ØØ CØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  82Ø
Ø77Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  77Ø

Ø78Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  78Ø
Ø79Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  79Ø
Ø7AØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  7AØ
Ø7BØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  7BØ
Ø7CØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  7CØ
Ø7DØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  7DØ
Ø7EØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  7EØ
Ø7FØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  7FØ

Ø8ØØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ 3A E1 F2 21 E4  B12
Ø81Ø  Ø8 BE 2Ø EE E1 DD 21 E4 Ø8 3A F9 Ø8 B7 3A E1 F2  ØAE
Ø82Ø  2A 41 F2 CØ BD 2Ø ØE 25 C8 95 2Ø Ø9 6F 67 22 3F  EØA
Ø83Ø  F2 2D 22 41 F2 3E FF 32 46 F2 CD FA 45 2B 36 ØØ  FB8
Ø84Ø  23 18 ØC 3A E4 Ø8 DD 96 ØØ 2Ø BA 32 FF F2 F1 CD  FDB
Ø85Ø  6B F3 E5 D5 3A E2 Ø8 4F 3A E3 Ø8 83 5F C5 7B Ø7  Ø29
Ø86Ø  Ø7 Ø7 E6 Ø7 ED 79 3E 8E ED 79 AF 47 ED 79 7B 17  FE1
Ø87Ø  E6 3E ED 79 3A E1 Ø8 4F E3 E3 ED B2 Ø6 ØØ ED B2  176

Ø88Ø  C1 1C 1Ø D9 ED 41 3E 8E ED 79 CD 68 F3 D1 E1 B7  237
Ø89Ø  C9 3A E4 Ø8 DD 96 ØØ C2 Ø8 Ø8 3C 32 FF F2 F1 CD  ØE1
Ø8AØ  6B F3 E5 D5 3A E2 Ø8 4F 3A E3 Ø8 83 5F 5Ø 7B Ø7  ØØ4
Ø8BØ  Ø7 Ø7 E6 Ø7 ED 79 3E 8E ED 79 AF 47 ED 79 7B 17  Ø31
Ø8CØ  E6 3E F6 4Ø ED 79 ØD E3 E3 ED B3 Ø6 ØØ ED B3 ØC  1A5
Ø8DØ  1C 15 2Ø DA ED 51 3E 8E ED 79 CD 68 F3 D1 E1 B7  1FC
Ø8EØ  C9 98 99 4Ø Ø2 FE ØØ Ø2 ØF Ø4 Ø1 Ø2 ØØ ØØ Ø1 3Ø  C63
Ø8FØ  Ø4 ØØ 5F ØØ Ø1 Ø1 ØØ FA Ø8 ØØ FE FF FF ØØ ED B2  EF2
```

リストA.1 "RAMDSK.X"

```
#include <stdio.h>
main(argc, argv)
int     argc;
char    **argv;
{
        FILE        *rfp, *wfp;
        unsigned    ad, sum, data, i, error;

        if (argc != 3) {
                fprintf(stderr, "Usage: pmud hex_file com_file¥n");
                return;
        }
        if ((rfp = fopen(argv[1], "r")) == NULL) {
                fprintf(stderr, "Cannot read '%s'¥n", argv[1]);
                return;
        }
        if ((wfp = fopen(argv[2], "wb")) == NULL) {
                fprintf(stderr, "Cannot write '%s'¥n", argv[2]);
                return;
        }

        error = Ø;
        while (fscanf(rfp, "%x", &ad) != EOF) {
                sum = ad;
                for (i = Ø; i <= 15; ++i) {
                        fscanf(rfp, "%x", &data);
                        putc((char)data, wfp);
                        sum += data;
                }
                fscanf(rfp, "%x", &data);
                if ((sum & ØxØfff) != data) {
                        printf("Error at line:%Ø4x¥n", ad);
                        error = 1;
                }
        }
        if (error == 1)
                unlink(argv[2]);
}
```

リスト A.2 〝MAKECOM.C〞

```
Ø1ØØ  11 8Ø ØØ 13 1A FE ØD C8 FE 2Ø 28 F7 FE Ø9 28 F3  7FØ
Ø11Ø  F3 21 FØ FB 22 FA F3 Ø6 28 1A 13 FE ØD 28 ØA FE  8B4
Ø12Ø  3B 2Ø Ø2 3E ØD 77 23 1Ø FØ 22 F8 F3 FB C9 ØØ ØØ  733
```

リスト A.3 〝AUTOKEY.X〞

索 引

マ

ヤ

ラ

■ 参考文献および推薦書籍

本書を執筆するにあたって以下の文献を参考にさせていただきました.

- MSX-C Ver.1.1　ユーザーズマニュアル　アスキー
- MSX-C Ver.1.2　ユーザーズマニュアル　アスキー
- C ハンドブック　石田晴久　訳　共立出版
- MS-C ハンドブック　アスキー書籍編集部編著　アスキー出版局
- プログラミング言語 C　B.W.カーニハン　D.M.リッチー　共著
石田晴久　訳　共立出版

本書を読み終えた方々のための次なるステップとして，以下の書籍をおすすめします.

- C プログラムブック I　打越浩幸　濱野尚人　梅原系　共著
- 入門 C 言語　三田典玄　著
- 実習 C 言語　三田典玄　著
- 改訂新版 C 言語入門　L.ハンコック　M.クリーガー　共著
アスキー出版局監訳

以上 4 点いずれもアスキー出版局

プログラム協力　薩山哲也

MSX-C 入門　上巻

1989年7月11日　初版発行
定価1,450円(本体1,408円)

著　者　桜田幸嗣
発行者　塚本慶一郎
発行所　株式会社アスキー
　〒107-24　東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
TEL (03)486-7111 (大代表)
情報 TEL (03)498-0299 (ダイヤルイン)
出版営業部 TEL (03)486-1977 (ダイヤルイン)

制　作　株式会社GARO
印　刷　凸版印刷株式会社

編　集　　　佐藤英一・竹内充彦
本文デザイン　川戸明子

ISBN4-7561-0006-6 C3055 P1450E